MASTER EN SCIENCES INFORMATIQUES À FINALITÉ SPÉCIALISÉE EN SOFTWARE ENGINEERING

Prise de décision répartie sur un réseau de moteur proactifs communicants

Andreux, Joprdan

Award date:
2018

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2017–2018

Prise de décision répartie sur un réseau de moteurs proactifs communicants

Jordan Andreux

Maître de stage : Denis Zampunieris

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Jean-Noël Colin

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

De nos jours, les systèmes d'information sont de plus en plus complexes et interconnectés. IBM et Intel proposent des solutions pour gérer ces environnements de manière autonome. Un moteur proactif permet de contrôler un système auquel il est connecté et de réagir à des évènements en temps réel. Cependant, l'interconnexion des systèmes rend le déploiement de plusieurs moteurs inévitable. Ce mémoire présente des pistes de solutions permettant de mettre en réseau plusieurs moteurs proactifs contrôlant chacun un système différent, mais également la mise en place d'une prise de décision répartie entre ces derniers.

Mots clés : Proactive computing, Algorithmes distribués, Communication, Prise de décision répartie

Abstract

Nowadays, information systems are increasingly complex and interconnected. IBM and Intel offer solutions to manage these environments independently. A proactive engine allows to control a system to which it is connected, and react to events in real time. However, the interconnection of systems makes the deployment of multiple engines unavoidable. This master thesis presents possible solutions for networking several proactive engines, each controlling a different system, but also for setting up a decision-making process shared between them.

Keywords : Proactive computing, Distributed algorithms, Communication, Distributed decision making

Avant-propos

Ce mémoire est le fruit de 4 mois de travail effectué lors de mon stage à l'Université du Luxembourg et de nombreuses heures d'analyse mais également de recherche. Celles-ci n'auraient pas pu être possibles sans ces personnes que je tiens tout particulièrement à remercier.

Le travail sur lequel se base la rédaction de ce mémoire a été accompli à l'Université du Luxembourg et plus particulièrement au sein de l'équipe de M. Zampunieris, mon maître de stage durant mon séjour. Je tiens tout particulièrement à le remercier pour son accompagnement, ses idées et son expertise qu'il a pu me fournir pendant toute la durée de mon stage.

Je tiens également à remercier toute son équipe et particulièrement Sandro Reis pour sa disponibilité et ses explications quant au fonctionnement du moteur proactif.

Mes remerciements s'adressent aussi à mon promoteur, Jean-Noël Colin, pour sa disponibilité et ses bons conseils qui ont pu m'être utiles pour la rédaction de ce mémoire.

Mes remerciements vont également à Daniel Chavée pour son temps et sa patience quant à la relecture de ce document.

Pour terminer, je souhaite remercier ma famille et ma compagne pour m'avoir accompagné et soutenu pendant ces derniers mois.

Table des matières

Table des figures

Chapitre 1

Introduction

De nos jours, l'informatique prend de plus en plus de place dans notre société et se répand dans des domaines qui étaient autrefois dépourvus de toute informatisation ou qui n'existaient pas. Cette expansion nous mène à des systèmes plus complexes souvent interconnectés à d'autres systèmes se trouvant à des endroits différents. L'expansion des moyens de communication n'y est pas étrangère, car ces derniers sont toujours plus fiables et diversifiés.

Deux paradigmes, l'« autonomic computing » et le « proactive computing » venant respectivement d'IBM et d'Intel, ont été pensés afin de réduire la complexité grandissante. Ces derniers permettent à un système de s'abstraire de l'interaction humaine en le rendant capable de se configurer, de s'optimiser et de se protéger automatiquement.

L'équipe de Denis Zampunieris provenant de l'Université du Luxembourg a mis au point un moteur proactif exécutant des scénarios sans action nécessaire de la part de l'utilisateur. Ce moteur a été testé en premier lieu sur la plateforme *Moodle* utilisée par l'Université. Connecté au système, le moteur était capable de gérer toute la partie administrative : rappel de remise d'un travail, changement d'horaire, etc. Les scénarios proactifs configurés étaient exécutés sans attente et sans action externe sur le système.

L'interconnexion des systèmes entre eux pousse le moteur proactif à s'adapter. Ce dernier est conçu pour être connecté et surveiller un seul système. De nos jours, un système ne compose plus nécessairement d'une seule unité centrale, il peut comporter plusieurs processus en réseau qui s'échangent de l'information mutuellement.

Le stage que j'ai effectué à l'Université du Luxembourg avait comme objectif de se concentrer sur le passage d'un environnement comportant un seul moteur connecté à un seul système vers un environnement comportant

plusieurs systèmes et nécessitant la mise en réseau de plusieurs moteurs proactifs.

L'objectif a pu être axé autour de trois questions de recherche : « Quels sont les scénarios locaux à développer pour chaque moteur ? », « Quel moyen de communication préférer entre les moteurs ? », « Quelle politique de management des moteurs mettre en œuvre afin de détecter, résoudre et prévenir les conflits potentiels pouvant survenir ? ».

Le travail de recherche a permis l'analyse de multiples moyens de communication existants pour de tels systèmes ainsi que leur évaluation permettant d'ajouter une dimension « réseau » au moteur proactif précédemment développé tout en conservant son architecture existante. Deux méthodes de communication ont été envisagées, l'envoi de message et la mémoire partagée.

La première partie de ce mémoire présente les deux moyens de communication pris en compte et les étapes par lesquelles il a été nécessaire de passer afin d'en mettre un des deux en place.

Dans la deuxième partie, nous verrons que possédant désormais une communication entre eux, les moteurs proactifs sont plus enclins à générer des conflits entre leurs décisions comme ils sont connectés à des (sous-)systèmes différents.

Pour pallier ce problème potentiel, une étude a été menée concernant la mise en place d'une politique de management entre ces moteurs proactifs. Les recherches effectuées ont fait ressortir deux solutions utilisées majoritairement dans les systèmes distribués : l'élection d'un *leader* et l'atteinte d'un consensus.

La dernière partie a été consacrée à l'application d'un réseau de moteurs proactifs dans des solutions réelles qui émergent du contexte de l'Internet des Objets.

Tous ces éléments nous ont conduits à l'implémentation d'un réseau de plusieurs moteurs proactifs capables de contrôler des systèmes distincts tout en intégrant une politique de management capable de gérer les conflits pouvant se produire.

Chapitre 2

Proactive Computing

2.1 Contexte

Depuis toujours, dans le domaine informatique, les chercheurs ont souvent axé leurs recherches sur l'informatique interactive. Cette démarche a pour but de rendre les interfaces toujours plus compréhensibles et accessibles au plus grand nombre de personnes. Un travail conséquent a été effectué sur l'interaction avec les utilisateurs tant au niveau interface graphique (UI/UX) qu'au niveau interaction homme-machine (souris, clavier ou modalités plus complexes).
Bien que cette branche de la recherche reste nécessaire à la conception de nouveaux logiciels, le nombre de machines par personne est en constante augmentation. Cette hausse est surtout due à la progression fulgurante des objets connectés en tout genre. L'humain se voit donc confronté à une limite indéniable, il ne pourra bientôt plus contrôler tous ses appareils électroniques par le biais d'une interface vu leur nombre grandissant.

C'est ce problème majeur que cherche à résoudre l'informatique proactive. La définition du terme proactif nous en dit plus sur le sujet : le mot vient de deux termes « pro » et « actif » et signifie anticiper les réponses à un problème, agir en se souciant des actions futures ou encore réagir à un fait postérieur. Ce terme est opposé à celui de « réactif » ou encore « passif » qui eux signifient « agir aux situations qui se présentent sur le moment » sans utiliser ce facteur de « prévision ».
En rendant les ordinateurs proactifs au lieu de réactifs, il est possible de sortir l'humain de la boucle d'interaction homme-machine. Les ordinateurs n'ayant plus de besoins d'interaction pour fonctionner, ils peuvent s'exécuter sans interruption 24h/24 et même 7j/7. En ajoutant à un système cette dimension proactive, il sera capable de réagir face à un évènement qu'il détecte en décidant lui-même quelle action exécuter. L'humain n'ayant plus ou presque plus d'interaction avec le système, ce dernier devient partiellement

voire entièrement autonome.
Ce concept est le proactive computing, il a été décrit pour la toute première fois par David Tennenhouse dans un article paru lors de l'année 2000 [15].

Pendant les 40 dernières années, la plupart des communautés de recherche en informatique se sont concentrées sur l'informatique interactive, guidée par la vision de J.C.R Licklider, centrée sur la symbiose entre l'humain et l'ordinateur [9].
Simultanément à cette recherche, l'industrie des technologies de l'information fonçait tout droit vers le point de rupture homme/machine/réseau, il s'agit du point où le nombre de machines interactives connectées sur un réseau surpassera le nombre de personnes sur terre.

La communauté de recherche en sciences informatiques savoure désormais une opportunité rare et excitante de redéfinir son agenda et d'établir les nouveaux objectifs qui vont propulser la société au-delà de l'informatique interactive et du point de rupture homme/machine/réseau. En se dirigeant vers un monde dans lequel les ordinateurs en réseau seront plus nombreux que les humains, nous devrions considérer ce que fera cet excès d'ordinateurs et élaborer un agenda de recherche tendant vers une augmentation de la productivité et de la qualité de vie humaine.

En réaction à la croissance des systèmes informatiques, une nouvelle vue des systèmes informatiques s'impose petit à petit. Quant à cette nouvelle vision, deux écoles sont opposées, celles d'IBM et d'Intel [16]. Au sein de chacune d'entre elles, la vision est assez différente, les choses présentées diffèrent en de nombreux points.

La première vision est celle d'IBM avec l'« autonomic computing », qui se traduit par « informatique autonome ». Elle consiste à maintenir une certaine connaissance de tous ses composants en toute circonstance. Grâce à celle-ci, le système peut jouir d'une dimension autonome de par sa résistance aux défaillances et sa capacité d'adaptation en cas de défaillance imprévue [7].

De l'autre côté, Tennenhouse (Intel) propose le « proactive computing », qui se traduit par « informatique proactive ». En opposition à IBM, cette vision propose qu'un système informatique puisse décider sur un évènement quelconque en fonction de son contexte actuel, ce qui ajoute une capacité d'anticipation par rapport aux évènements. Dans son article, il décrit 3 caractéristiques de cette nouvelle activité de recherche [15] :

- **Getting physical :** Les systèmes proactifs seront directement connectés au monde qui nous entoure en utilisant des capteurs et des actuateurs

afin de surveiller et façonner leur environnement physique. La recherche dans ce point explore le couplage omniprésent des systèmes connectés à leur environnement.

- **Getting real :** Les ordinateurs proactifs vont répondre de façon routinière à des stimuli externes à une vitesse plus rapide que celle des humains. La recherche dans ce domaine doit combler le fossé entre la théorie du contrôle et l'informatique.
- **Getting out :** L'informatique interactive place délibérément l'homme dans la boucle d'interaction. Cependant, la réduction des constantes de temps implique des recherches sur les modes de fonctionnement proactifs dans lesquels l'homme est placé au-dessus de cette boucle.

Il y a deux raisons simples pour lesquelles nous devrions détourner certaines de nos ressources intellectuelles vers l'informatique proactive : la vaste majorité des nouveaux ordinateurs seront proactifs et ils seront la principale source d'informations.

Les deux visions ne s'excluent pas mutuellement, elles peuvent coexister ou se compléter. Cependant, l'informatique doit être repensée, car les nouveaux systèmes informatiques ne seront pas seulement interactifs, mais ils seront aussi embarqués. Cette mobilité va permettre d'être en contact direct avec un environnement permettant le contrôle des évènements physiques qui se produisent. Cela implique un amas de données supplémentaires que l'humain ne peut traiter seul, il devra dédier cette tâche aux machines et se concentrer principalement sur la supervision de ces dernières.

2.2 Différentes visions

2.2.1 Autonomic computing

Un manifeste publié par IBM en 2001 montre la croissance du nombre de systèmes en circulation et surtout la difficulté de tous les maintenir en cas de défaillance de plusieurs d'entre eux [7]. Les systèmes d'entreprises sont un exemple d'une classe de systèmes complexes qui doit fonctionner constamment de manière fiable et consistante, et ce, même en l'absence d'implication humaine. De nos jours, la plupart des tâches de maintenance des systèmes ne peuvent plus être prises en charge de manière suffisamment efficace par des opérateurs manuels, même compétents.

Une solution proposée par IBM est l'« autonomic computing » (informatique autonome), il est décrit comme des systèmes informatiques qui peuvent gérer tout seuls des objectifs de haut niveau pour des administrateurs.
Quand le premier vice-président de la recherche chez IBM a introduit cette

idée pour la première fois, il a fait une analogie avec la biologie et le corps humain. Le système nerveux autonome gouverne notre fréquence cardiaque et la température de notre corps, libérant ainsi notre cerveau de la charge de ces fonctions et de bien d'autres de bas niveau, mais qui sont vitales.

L'essence de l'informatique autonome est le « self-management ». L'intention de cela est de libérer les administrateurs système des détails de ce qui touche aux opérations et à la maintenance d'un système. Ce dernier pourrait surveiller constamment sa propre utilisation et vérifier les mises à jour de ses composants par exemple. IBM cite fréquemment quatre aspects de self-management : le self-configuring présenté au point 2.2.1.1, le self-healing présenté au point 2.2.1.3, le self-optimizing et le self-protecting présentés respectivement aux points 2.2.1.2 et 2.2.1.4.
Le tableau ci-dessous présente également ces quatre aspects, mais sous forme d'une comparaison entre l'informatique telle qu'on la connaît actuellement et l'informatique autonome.

Table 1. Four aspects of self-management as they are now and would be with autonomic computing.

Concept	Current computing	Autonomic computing
Self-configuration	Corporate data centers have multiple vendors and platforms. Installing, configuring, and integrating systems is time consuming and error prone.	Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly.
Self-optimization	Systems have hundreds of manually set, nonlinear tuning parameters, and their number increases with each release.	Components and systems continually seek opportunities to improve their own performance and efficiency.
Self-healing	Problem determination in large, complex systems can take a team of programmers weeks.	System automatically detects, diagnoses, and repairs localized software and hardware problems.
Self-protection	Detection of and recovery from attacks and cascading failures is manual.	System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent systemwide failures.

FIGURE 2.1 – Les 4 aspects principaux de l'informatique autonome [7]

2.2.1.1 self-configuration

Installer, configurer et intégrer des systèmes larges et complexes met au défi, prend du temps, et est sujet aux erreurs, même pour les experts.

Les systèmes autonomes intègrent une dimension d'auto-configuration selon des politiques de haut niveau qui spécifient ce qui est voulu, et non comment cela doit être fait. Quand un composant est introduit, il va s'incorporer parfaitement et le reste du système va s'adapter à sa présence (à la

manière d'une nouvelle cellule au sein du corps ou d'une nouvelle personne dans la population).
En étant introduit, il s'enregistre lui-même ainsi que ses capacités de telle manière que les autres composants peuvent l'utiliser ou modifier son comportement de manière appropriée.

2.2.1.2 self-optimization

Les middlewares complexes peuvent avoir des centaines de paramètres réglables afin d'obtenir des performances optimales sur le logiciel.

Avec l'aspect d'auto-configuration, les systèmes vont continuellement chercher un moyen d'améliorer leurs opérations, identifier et saisir les opportunités qui va leur permettre d'être plus efficace en performance et en coût. Les systèmes autonomes seront capables de se surveiller, d'expérimenter avec, et de régler leurs paramètres et ils pourront apprendre afin de faire les choix adéquats à propos du maintien ou de la sous-traitance de leurs fonctions.

2.2.1.3 self-healing

La plupart des sociétés informatiques ont de larges départements dévoués à identifier, tracer et déterminer la cause d'erreurs dans des systèmes informatiques complexes.

L'aspect d'auto-réparation prévoit qu'un système sera en mesure de détecter, diagnostiquer et réparer des problèmes locaux résultant de bugs ou d'erreurs dans les logiciels et matériels. En utilisant la connaissance à propos de la configuration du système, un composant de diagnostic de problèmes analysera l'information à partir de logs, éventuellement complétés par des données provenant de moniteurs additionnels.
Le système devra ensuite comparer le diagnostic avec les corrections connues, installer cette correction et re-tester.

2.2.1.4 self-protecting

Malgré l'existence de nombreux firewalls et outils de détection d'intrusion, ce sont les hommes qui décident comment protéger les systèmes des attaques malicieuses ou d'erreurs en cascade.

Les systèmes autonomes seront auto-protégés sur deux grands axes. D'une part, ils se défendront dans leur ensemble contre des problèmes corrélés résultant d'attaques malveillantes ou d'échecs en cascade qui ne sont pas corrigés par les mesures d'auto-réparation. D'autre part, ils anticiperont les problèmes basés sur les rapports récents des capteurs et ils prendront des mesures afin de les éviter ou de les atténuer.

2.2.2 Proactive computing

Le concept de « proactive computing » (informatique proactive) est apparu pour la première fois en 2000 dans un article « Proactive Computing » tiré de la revue « Communications of the ACM » et écrit par David Tennenhouse [15]. Dans son article, Tennenhouse décrit que l'humain ne doit plus être placé dans la boucle d'interaction avec le système, mais au-dessus de celle-ci. En le retirant de la boucle, l'informatique proactive se concentre sur l'humain comme un superviseur, uniquement nécessaire pour fournir des conseils en cas de décisions critiques.
Le design des systèmes proactifs est guidé par sept principes [16] :

1. Être connecté au monde physique
2. Être interconnecté à plusieurs niveaux des réseaux
3. Gérer le macro-traitement
4. Accepter l'incertitude et l'imprévu
5. Anticiper
6. Répondre en temps réel au sein d'une boucle fermée
7. Rendre le système personnel

Ces sept principes sont regroupés au sein de trois catégories qui sont : « Getting physical », « Getting real », « Getting out » et qui ont déjà été développées dans la section 2.1.

2.2.3 Proactive et Autonomic computing

Comme montré à la figure 2.2, il y a un chevauchement intellectuel entre la recherche dans les systèmes proactifs et les systèmes autonomes. Les deux sont nécessaires pour nous fournir des outils afin de faire évoluer les systèmes informatiques vers un large éventail de nouveaux champs d'application. D'un côté, l'ajout d'autonomie cherche à résoudre certains problèmes dus à la croissance des applications informatiques devenant de plus en plus complexes. De l'autre côté, l'ajout de proactivité étend notre horizon en reconnaissant le besoin de contrôler et mettre en forme le monde physique, où les interactions avec le monde réel sont complexes et limitées par l'implication humaine.

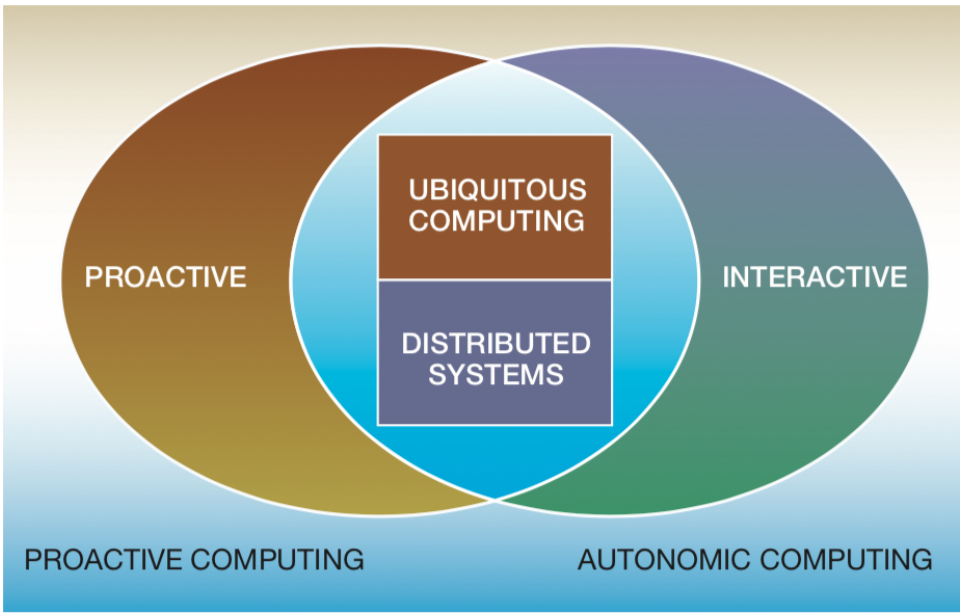

FIGURE 2.2 – La relation entre les paradigmes informatiques [16]

2.3 Implémentations existantes

Plusieurs implémentations des concepts du chapitre 2 existent. Cette section va présenter de manière succincte certaines solutions et d'autres vont être explorées plus en profondeur.

2.3.1 Le moteur de l'Université du Luxembourg

Au sein de l'unité de recherche en sciences informatiques à l'Université du Luxembourg, l'équipe de M. Zampunieris a développé un moteur proactif destiné à des besoins dans l'e-learning et plus précisément sur la plateforme « Moodle ».
C'est ce moteur qui a été utilisé afin d'entreprendre les recherches dans le cadre de ce mémoire.

Dans le premier article paru, un système d'e-learning proactif est proposé et décrit comme suit : « Un système de gestion d'apprentissage conçu pour aider ses utilisateurs à avoir une meilleure interaction en ligne dans un environnement éducatif ou d'apprentissage, en fournissant des analyses programmables, automatiques et continues des (inter-)actions des utilisateurs agrémenté d'actions appropriées initiées par le système proactif lui-même » [17].
Ce genre de système d'apprentissage n'est que réactif, basé sur une interface axée utilisateurs, il attend une action avant de réagir, ce qui montre

ses limitations et son inefficacité dans le temps de réaction aux requêtes utilisateurs [18].

2.3.1.1 Fonctionnement

Le moteur est un système expert dynamique basé sur l'exécution de règles. Chaque comportement proactif est donc codé au sein d'une règle et l'association de plusieurs règles permet de créer le concept (abstrait) de scénario.
C'est par itération que les règles sont ensuite exécutées les unes après les autres en respectant le principe de la file FIFO.

2.3.1.2 Composants

Tout d'abord, afin de stocker les différentes règles à exécuter, le moteur possède deux files distinctes.

- **Current queue :** il s'agit de la file "actuelle". Lors d'une itération, le moteur exécute une à une les règles se trouvant dans cette file-ci.
- **Next queue :** il s'agit de la file "suivante". Lors d'une itération, d'autres règles peuvent être créées. Ces règles seront stockées dans cette file et exécutées lors de l'itération suivante.

Une itération touche à sa fin lorsque la « current queue » ne possède plus de règles à exécuter, elle est vide. À ce moment, le moteur prépare l'itération suivante en copiant les règles de « next queue » dans « current queue » tout en préservant l'ordre. Après, il affecte « next queue » à « null » afin de la vider de son contenu.

Ensuite, le moteur permet la paramétrisation de trois paramètres différents :

- **F :** il s'agit de la fréquence d'une itération (exprimée en millisecondes).
- **N :** il s'agit du nombre de règles maximum qu'il est possible d'exécuter au sein d'une itération.
- **P :** il s'agit du temps de pause minimum (« minimum pause ») entre deux itérations.

Le moteur possède également une base de données qui lui est propre et qui lui permet de sauvegarder les règles de « current queue » avant le début d'une itération. Ce mécanisme permet la récupération des règles qui allaient être exécutées lors de l'arrêt inopiné du moteur.

2.3.1.3 Les règles

Une règle est une classe Java, respectant une structure commune à toutes et qui implémente une interface définie. Cette classe peut posséder en plus des méthodes obligatoires contenues dans l'interface, un certain nombre d'attributs et de méthodes supplémentaires.

```
protected void dataAcquisition() ;
```

1. **Data acquisition :** Il s'agit de la première partie d'exécution d'une règle. C'est ici que toutes les données sont récupérées comme les éventuels paramètres de la règle, ce qui provient de la base de données grâce au « wrapper », des éventuels calculs sur les données, etc. Tout ce qui doit être acquis pour assurer la bonne exécution de la règle s'obtient dans cette partie.

```
protected boolean activationGuards() ;
```

2. **Activation guards :** C'est une suite de conditions logiques qui doivent être satisfaites afin de continuer l'exécution de la règle. Le résultat de la condition est stocké au sein d'une variable locale appelée « activated » et la valeur de celle-ci détermine la suite du flux d'exécution. Si la variable contient « vraie », la règle exécutera la méthode n°3 ; sinon, elle exécutera directement la méthode n°5 appelée quelque soit le parcours au sein du flux.

```
protected boolean conditions() ;
```

3. **Conditions :** Il s'agit d'une condition supplémentaire afin d'exécuter la méthode n°4. Cette méthode est également une suite de conditions logiques, mais regroupant généralement des tests plus complexes qu'il n'est pas nécessaire de calculer dans la méthode n°2. Si cette condition n'est pas satisfaite, le flux d'exécution passe directement à l'exécution de la méthode n°5.

```
protected void actions() ;
```

4. **Actions :** Cette méthode est le corps de la règle. C'est ici qu'une règle effectue ses actions pour lesquelles elle est programmée. Pour que son exécution se déroule comme prévu, les méthodes n°2 et n°3 doivent être vérifiées et les données de la méthode n°1 doivent être récupérées correctement.

```
protected boolean rulesGeneration() ;
```

5. **Rules generation :** Quoiqu'il se passe dans les autres méthodes, celle-ci est exécutée. C'est dans cette partie qu'une règle peut en générer

d'autres ou se cloner elle-même afin de continuer l'exécution des scénarios.

```
i.    repeat for each data acquisition request DA
        a. perform DA
        b. if error then raise exception on system manager console and go to step vii
           else create new local variable and initialize it with result of DA
ii.   create new local Boolean variable "activated" initialized to false
iii.  repeat for each activation guard test AG
        a. evaluate AG
        b. if result == false then go to step vi
           else     if AG == last activation guard test
                    then activated = true
iv.   repeat for each conditions test C
        a. evaluate C
        b. if result == false then go to step vi
v.    repeat for each action instruction A
        a. perform A
        b. if error then raise exception on system manager console and go to step vii
vi.   repeat for each rule generation R
        a. perform R
        b. insert newly generated rule as the last rule of the system
vii.  delete all local variables
viii. discard rule from the system
```

FIGURE 2.3 – Algorithme d'exécution d'une règle au sein du moteur [17]

2.3.1.4 Les scénarios

Les règles ayant été définies dans la section 2.3.1.3, il est possible de définir le concept de scénario. Le moteur proactif ne connaît pas cet élément, il ne s'agit que d'un concept abstrait permettant à l'utilisateur de se représenter un flux de données au sein de celui-ci.
Un scénario est composé d'une suite de règles et est orienté vers un but bien précis. Chacun d'entre eux possède un but bien défini et leurs règles contiennent les étapes successives à accomplir afin de l'atteindre.
Il existe deux types de scénarios bien distincts [13] :

Meta Scenarios

Le but de ce type de scénario est de fournir au système une fonction axée sur la perception, c'est-à-dire de détecter un évènement et entreprendre les actions appropriées [13]. La fonctionnalité principale de ce type de scénario est d'être une règle consciente de l'environnement du moteur à tout moment,

car elle ne se termine jamais. Dès qu'un évènement correspondant est détecté par le Meta Scenario, il déclenche le Target Scenario correspondant afin de réagir en conséquence.
Le Meta Scenario ne réagit pas face aux évènements, il ne fait que les détecter pour ensuite déléguer le job au Target Scenario approprié.

Target Scenarios

Le but de ce type de scénario est de fournir des réponses cibles multiples pour chaque évènement (ou non-évènement) détecté par un Meta Scenario [13]. Ce sont ces scénarios qui effectuent les actions initiées par les Meta Scenarios, mais contrairement à eux, une fois le travail effectué, chaque règle du Target Scenario est éliminée.
Il s'agit de la principale différence entre les deux, un Meta Scenario ne disparaît jamais du moteur proactif, car sa présence est permanente et un Target Scenario disparaît entièrement une fois que son action pour laquelle il est prévu est effectuée.
Cette approche permet de garder au sein du moteur uniquement les règles nécessaires à son bon fonctionnement et permet également une certaine optimisation.

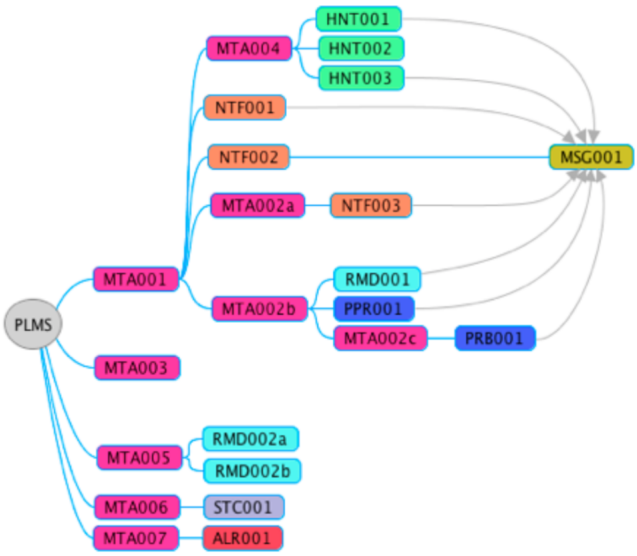

FIGURE 2.4 – Représentation générique de scénarios [13]

.

2.3.1.5 Les files

Comme détaillé à la section 2.3.1.2, le moteur proactif possède deux files : « currentQueue » et « nextQueue ».

La file actuelle contient toutes les règles qui seront exécutées à l'itération actuelle.
Dans la file suivante se trouvent les règles qui seront exécutées à l'itération suivante. Les règles contenues dans cette file sont celles qui sont générées par les règles de la file actuelle lorsque leur méthode *rulesGeneration()* est appelée.

2.3.1.6 Itérations

Une itération est le flux d'exécution du moteur. A la section 2.3.1.2, trois paramètres sont définis et peuvent être ajustés afin de paramétrer une itération. Lors de l'exécution de celle-ci, les règles au sein de la file « currentQueue » sont exécutées les unes après les autres selon leur ordre dans celle-ci. Ces règles peuvent effectuer des actions et/ou générer de nouvelles règles.
Une itération se termine lors que la file courante est vide.

2.3.1.7 Exécution

Grâce aux composants détaillés, il est possible de décrire une exécution complète du moteur.

Avant le début de l'itération, le moteur sauvegarde ses règles dans sa base de données locale à des fins de restauration. Ensuite, le moteur commence par exécuter les règles de la file actuelle selon leur ordre d'apparition. Chacune de celles-ci peut effectuer des actions et/ou générer de nouvelles règles (y compris elle-même) qui seront stockées dans « nextQueue ». L'itération poursuit son exécution tant que la file actuelle n'est pas vide. À la fin de l'itération, le moteur enregistre quelques statistiques sur celle-ci.

Une fois l'itération terminée, la file actuelle est vide puisque toutes ses règles ont été exécutées. À partir d'ici, deux options s'offrent au moteur.
Si la file suivante n'est pas vide, son contenu est transféré dans la file courante tout en préservant son ordre. La file courante contient donc les règles de la file suivante qui devient elle-même vide. L'itération suivante peut alors commencer en exécutant de nouveau les règles de « currentQueue ».
Dans l'autre cas, si « nextQueue » est vide, le moteur n'a rien à exécuter et stoppe son exécution.

Ce n'est pas l'unique façon pour le moteur de terminer son exécution, il peut aussi recevoir un signal par sa base de données. Si, à la fin d'une itération, il a le « flag » d'arrêt dans la base de données, le moteur arrête son exécution et sauvegarde les règles encore présentes pour une exécution future.

En ce qui concerne l'accès aux données pour le moteur, il possède deux technologies. La première est la mise à disposition d'un « wrapper » qui permet de se connecter à un système distant (comme la base de données du système qu'il surveille par exemple). C'est ici que le moteur va chercher l'environnement sur lequel se baser et sur lequel travailler.
La deuxième est sa base de données locale. Le moteur est codé en Java et possède une librairie nommée Hibernate permettant d'accéder ou de stocker des données localement. Cette base de données permet de sauvegarder les règles, les statistiques sur l'exécution, les paramètres d'itération ou encore d'autres informations nécessaires au fonctionnement du moteur ou à l'exécution des règles.

2.3.2 Améliorations du moteur proactif

La conception du moteur proactif tel que décrit à la section 2.3.1 n'en est pas restée là. À des fins d'optimisations, l'exécution du moteur a été améliorée et il a également été porté sur smartphone.

2.3.2.1 Exécution asynchrone

Lors de l'année académique 2016-2017, Florian Daloze, un étudiant de l'Université de Namur a travaillé sur la modification de l'architecture et le fonctionnement du moteur proactif de l'Université du Luxembourg [5].

Le moteur détaillé à la section 2.3.1 possède une architecture qui rend son exécution synchrone. À chaque itération, les règles sont exécutées une à une. Une fois le moteur dépourvu de toute règle, son exécution se termine et il s'arrête.

Dans l'amélioration proposée par Florian, le moteur passe d'une exécution synchrone à une exécution asynchrone.

Une des principales avancées par rapport au moteur de la section 2.3.1 est la suppression des itérations qui sont remplacées par un unique pool de *threads*. C'est ce dernier qui permet l'exécution asynchrone. Le moteur possède un *thread* principal qui gère les arrivées de règles et qui les dispatche dans le pool de *threads*.

Chaque règle possède désormais son propre paramètre p pour permettre un délai entre chaque exécution d'une règle et est capable de s'exécuter dans un *thread* au sein du pool de *threads*.

L'exécution étant asynchrone, il est nécessaire de verrouiller les accès multiples aux ressources. Pour implémenter ceci, chaque ressource dispose de verrous que la règle doit posséder afin d'y accéder, si elle le souhaite.

2.3.2.2 Moteur embarqué

Une autre version du moteur existe, il s'agit d'une version « embarquée » de celui-ci. Lors de l'année académique 2015-2016, un étudiant de l'Université du Luxembourg a travaillé sur la portabilité du moteur proactif existant sur un smartphone Android [11].

Le moteur proactif de la section 2.3.1 a été adapté et porté sur un smartphone tournant sur un environnement Android.

Par la suite, en plus de cette adaptation, une communication entre différents moteurs proactifs (embarqués ou non) a été réfléchie et analysée [12].

2.3.3 Autres solutions existantes

Le moteur proactif de l'Université du Luxembourg n'est pas l'unique solution qui existe à ce jour.

2.3.3.1 PureStorage

La société PureStorage propose des solutions stockage. Leur infrastructure est pourvue d'une détection proactive des erreurs afin de prendre les mesures préventives nécessaires avant qu'un problème ne survienne.
Le résultat est que le risque d'interruption est limité, car les clients sont prévenus des problèmes et les correctifs sont également planifiés [2].

Cette détection prédictive et une réponse proactive permettent de :

— Prédire les vulnérabilités des problèmes connus.
— Détecter les changements de configuration et environnementaux en temps réel.
— Détecter les anomalies de performance et de capacité.

2.3.3.2 Pre Safe Assist

Volkswagen possède au sein de ses véhicules un système d'assistance proactif. Lorsque le véhicule se trouve dans une situation critique, le système embarqué le détecte et adapte l'environnement du véhicule afin d'éviter la gravité d'un accident qui pourrait se produire [1].

Prenons l'exemple de Volkswagen, dans un virage avec une flaque d'eau, la voiture dérape et le système le détecte. Il réagit comme suit :

— Les ceintures de sécurité à l'avant se resserrent.
— Les vitres se referment avant d'assurer une protection optimale avec les airbags de tête.
— L'éventuel toit ouvrant se referme, s'il est ouvert.

À la fin du virage, si tout s'est bien passé, tout redevient comme avant afin de retrouver l'état initial.

Chapitre 3

Algorithmes distribués

3.1 Contexte

Le concept d'*algorithmes distribués*, parfois appelé *algorithmes répartis*, couvre une large variété d'algorithmes concurrents utilisés dans un large éventail d'applications.

À l'origine, ce terme était utilisé pour représenter des algorithmes conçus pour être exécutés sur des processeurs « distribués » dans une zone géographique assez large. Mais avec le temps, l'usage de ce terme s'est élargi et inclut désormais des algorithmes s'exécutant sur un réseau local et même des algorithmes pour des multiprocesseurs à mémoire partagée. En effet, tous ces algorithmes utilisés dans des contextes différents ont beaucoup de paramètres en commun.

Les algorithmes distribués sont utilisés dans de nombreux contextes. Parmi eux, nous pouvons en citer quelques-uns comme les télécommunications, le calcul scientifique ou encore le contrôle des processeurs en temps réel. Dans le contexte des télécommunications, l'algorithme de Dijkstra [6] et celui de Bellman-Ford [4] sont les plus connus.

En plus des différents contextes, les algorithmes distribués peuvent prendre plusieurs formes et les attributs par lesquels ils se différencient comprennent un ou plusieurs points suivants [10] :

- **La méthode de communication interprocessus :** Les algorithmes distribués s'exécutent sur une collection de processeurs, qui doivent communiquer d'une manière ou d'une autre. Cette communication peut prendre la forme d'une mémoire partagée commune, envoyer des messages point-à-point ou les diffuser au sein de tout le réseau.
- **Le modèle de synchronisation :** Plusieurs hypothèses peuvent être faites

au sujet de la chronologie des évènements dans un système. À un extrême, les processus peuvent être complètement synchrones avec des verrous et la synchronisation. À l'autre extrême, les processus peuvent être complètement asynchrones s'exécutant à des vitesses et dans des ordres arbitraires. Entre les deux extrêmes, il existe des processus pouvant être regroupés sous la dénomination de partiellement synchrones.

- **Le modèle de défaillance :** Le matériel sur lequel les processus s'exécutent peut ne pas s'avérer complètement fiable ou les algorithmes doivent avoir besoin de tolérer un certain seuil de comportement défaillant. Ce comportement peut prendre plusieurs formes comme des processus s'arrêtant sans alerte (*stopping failures*) ou des processus exposant des valeurs arbitraires (*Byzantine failures*). Un comportement peut aussi inclure des défaillances du mécanisme de communication.
- **Les types de problèmes :** Les algorithmes diffèrent également dans le type de problème qu'ils doivent résoudre. Cela comprend des problèmes d'allocation des ressources, de communication, de consensus entre les processus, de synchronisation, etc.

Dans l'algorithmique distribuée, il est important de ne pas confondre les systèmes distribués et les systèmes parallèles.
Un système parallèle consiste à séparer un gros problème en problèmes plus petits permettant de les traiter en parallèle et d'exécuter le plus d'opérations possible dans un intervalle de temps donné.
Un système distribué consiste à faire interagir et coopérer des processus indépendants afin de réaliser une tâche donnée.

Cette section s'intéresse principalement aux systèmes distribués, leurs moyens de communication, leurs synchronisations et leurs défaillances possibles et les types de problèmes qu'il est possible de résoudre par le biais de ces systèmes.

3.2 Méthodes de communication

La première caractéristique d'un système distribué est le moyen de communication que celui-ci utilise. Elle peut prendre différentes formes en fonction des besoins et de la capacité du réseau.

Cependant, il est possible de les regrouper en deux catégories : Communication par échange (passage) de messages ou communication par variable (mémoire) partagée [14].

Malgré les deux paradigmes, il existe un résultat bien connu permettant de simuler une communication par échange de messages par une communication via mémoire partagée et vice-versa. Ce résultat rend en quelque sorte les deux paradigmes assez équivalents [8].

3.2.1 Échange de messages

La première catégorie concernant les moyens de communication est l'échange de messages entre processus. En reprenant la différence entre un système distribué et parallèle, on peut voir que le but d'un système distribué est de faire coopérer et interagir des processus indépendants entre eux. Pour effectuer ces actions, il est nécessaire d'avoir une couche de communication entre ceux-ci.

Un système distribué peut être modélisé par un graphe fortement connecté. Ce graphe représente le réseau en entier où chaque nœud est un processus et chaque arc est un lien de communication entre processus.

Chaque processus est connecté à tous les autres nœuds du réseau et peut leur envoyer un message. Cette communication peut prendre plusieurs formes.

Soit il s'agit d'un échange de messages point-à-point (*end-to-end*) où un processus envoie un message à un autre processus auquel il est connecté directement sans informer les autres nœuds du réseau de cet envoi de message. Cet envoi de messages entre des nœuds du réseau est modélisé à la figure 3.1.

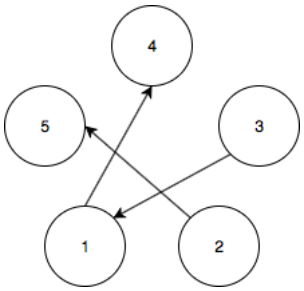

FIGURE 3.1 – Envoi de messages point-à-point

Soit il s'agit d'un échange de messages en diffusion (*broadcast*). Un nœud souhaite envoyer un message, mais au lieu de l'envoyer un autre nœud directement, celui-ci le diffuse à tous les nœuds du réseau y compris lui-même. Il existe une variante qui est le *unicast* où un nœud diffuse un message vers tous les nœuds du réseau, mais sans s'inclure dedans. Cet envoi de message

(*broadcast*) dans le réseau est modélisé à la figure 3.2, l'*unicast* est modélisé de la même façon en enlevant la flèche réflexive sur le nœud 1.

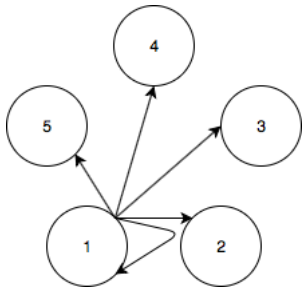

FIGURE 3.2 – Envoi de messages en *broadcast*

Dans un modèle de communication par échange de messages, chaque processus connaît l'information qu'il a recueillie localement. Pour connaître l'information de ses voisins, il doit communiquer en échangeant cette information dans des messages.

Pour échanger cette information locale, chaque processus possède deux opérations *send()* et *receive()* qui respectivement permettent d'envoyer et recevoir les données.

La complexité de ce moyen de communication se situe au niveau du nombre de messages envoyés par un processus. Plus un processus envoie des messages à travers le réseau, plus son exécution est ralentie et prendra du temps.

3.2.2 Mémoire partagée

L'autre catégorie concernant les moyens de communication est radicalement différente de la précédente présentée à la section 3.2.1. Au lieu de posséder des canaux de communications entre les nœuds, ceux-ci sont directement connectés à une mémoire partagée qui permet de centraliser l'échange d'informations.

Chaque nœud est donc désormais connecté à une mémoire partagée qui centralise les données de chaque processus. Ils peuvent effectuer tant des opérations de lecture que des opérations d'écriture sur cette mémoire.

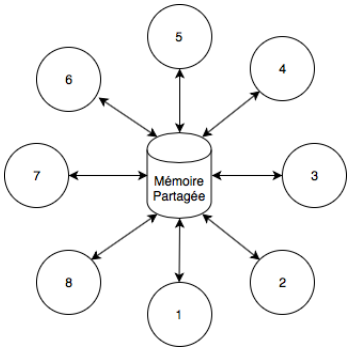

FIGURE 3.3 – Communication par une mémoire partagée

Dans un modèle de communication par mémoire partagée, chaque processus connaît l'information qu'il a recueillie localement, mais en plus de celle-ci il peut connaître l'information recueillie par ses voisins en accédant à la mémoire partagée.

Pour échanger cette information locale, chaque processus possède deux opérations *write()* et *read()* qui respectivement permettent d'écrire et lire les données de la mémoire partagée.

Dans ce modèle, comparé à celui de la section 3.2.1, la communication est fortement réduite. Il en résulte que la complexité ne se situe plus au niveau des messages, mais elle se situe désormais au niveau du nombre d'opérations en lecture/écriture d'un processus sur la mémoire. En effet, plus un processus accède à cette mémoire, plus son exécution globale est ralentie et elle prendra plus de temps.

3.3 Modèles de synchronisation

Abordons une autre caractéristique des systèmes distribués, le modèle de synchronisation. Chaque processus du réseau peut travailler à des vitesses différentes les unes des autres. Dès lors, il est nécessaire de considérer une dimension de synchronisation entre ces processus [10].

Il peut exister des modèles où, à chaque étape de l'algorithme, les processus envoient leurs messages de manière synchrone et puis continuent leur exécution ou bien les processus envoient leurs messages dès qu'ils peuvent et travaillent à des vitesses arbitraires et différentes ce qui ajoute une couche de complexité au système.
Certains systèmes ne font pas partie d'un des deux modèles « extrêmes »,

mais se situent entre les deux en étant partiellement synchrones.

Un modèle de synchronisation intervient sur deux éléments différents : l'exécution et la communication [8].

3.3.1 Modèle synchrone

Afin de comprendre plus en profondeur le modèle synchrone, il est nécessaire de détailler l'exécution dans ce modèle.

Une exécution dans un système distribué se divise en différents *rounds*. Pendant ces rounds, un processus peut effectuer deux types d'actions : faire des calculs locaux (*compute*) ou communiquer via le modèle implémenté (*send*/*receive*).

À chaque *round*, un processus effectue des calculs locaux (*compute*), il communique ses données calculées aux autres processus (*send*) et reçoit également des données des processus du système (*receive*). L'exécution d'un tel système est donc un enchaînement d'actions *compute, send, receive, compute, send, receive*, etc. On dit que le système est dirigé par les synchronisations d'horloge qui forme les *rounds*.

Afin de rendre cette exécution synchrone, chaque processus possède une horloge interne qui est synchronisée entre tous les processus. Lors de la communication entre processus, le délai de transmission est borné par une valeur qui est connue de tous les processus du système. Lors de la communication, le processus expéditeur « se bloque » jusqu'à attendre la bonne réception de son message au destinataire. Cette borne implique qu'aucun processus n'attendra indéfiniment la remise de son message et ne sera dès lors jamais bloqué dans son exécution [8].

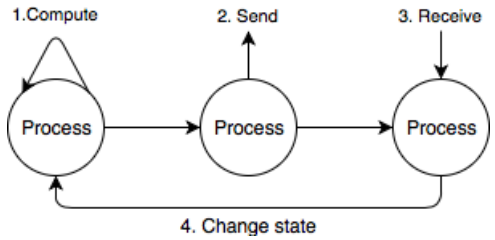

FIGURE 3.4 – Exécution synchrone d'un processus

Il est également possible de représenter les processus du système distribué comme des machines à états classiques. Mais, en plus de posséder une

fonction de transition d'un état à un autre, chaque processus possède une fonction de génération de messages qui correspond aux actions *send* et *receive*.

L'exécution d'un processus comme une machine à états est déterministe. À chaque tour, il n'y a qu'un seul envoi de message et état suivant possible. L'exécution d'un processus dans un système synchrone suit donc un seul et unique chemin [10].

3.3.2 Modèle asynchrone

En opposition au modèle synchrone de la section 3.3.1, les horloges des différents processus ne sont plus synchronisées si bien que celles-ci n'ont plus d'utilité. En effet, lors de l'échange d'un message d'un processus vers un autre, celui-ci arrive en un temps fini, mais imprédictible.

Contrairement au modèle synchrone qui est dirigé par les synchronisations d'horloge, un système distribué asynchrone est dirigé par la réception de messages. Dans un modèle synchrone où les processus lancent leurs actions à chaque « top » d'horloge, les processus d'un modèle asynchrone lancent leurs actions dès qu'ils reçoivent un message.

L'exécution se déroule comme ceci. Un processus effectue ses calculs locaux et ensuite il envoie un message à un ou plusieurs processus. Comme ces derniers ne sont pas synchronisés, le processus attend la réception d'un message afin de faire l'action associée à ce message. Les processus n'étant pas synchronisés, ils peuvent recevoir des messages qui leur sont impossibles à traiter en temps réel. Le message est simplement mis dans un *buffer* et est délivré au receveur dès que celui-ci est prêt à l'accepter [8].

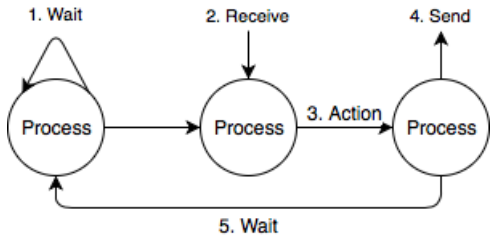

FIGURE 3.5 – Exécution asynchrone d'un processus

Ce modèle peut aussi être représenté par des machines à états comme le modèle synchrone. Cependant, l'exécution n'est plus déterministe, car il

n'existe pas un seul chemin dans le système. Les processus peuvent recevoir les messages, et par conséquent d'états, dans un ordre arbitraire. C'est ce qui fait la principale différence avec le modèle de la section 3.3.1 [10].

3.3.3 Modèle partiellement synchrone

Le modèle de synchronisation de cette section est plus réaliste que ses prédécesseurs (section 3.3.1 et 3.3.2), car les composants du système ont quelques informations sur le temps. Cependant, elles ne sont pas aussi complètes que dans le modèle synchrone.

L'exécution d'un modèle synchrone est divisée en différents *rounds*, ceci est possible grâce à une horloge interne synchronisée entre tous les processus. Le système est dirigé par les synchronisations d'horloge.
L'exécution d'un modèle synchrone n'est pas divisée en *rounds*, un processus traite les messages reçus « à la volée ». Le système est dirigé par la réception des messages.

Le modèle partiellement synchrone est une sorte de « fusion » des deux modèles précédents. Le système est également dirigé par les réceptions des messages, mais la réception de ceux-ci se fait en un temps borné [10].
La réception de messages en modèle asynchrone se termine en un temps fini, mais non prédictible. En modèle synchrone, elle se termine en un temps fini et borné et les processus sont synchronisés.

La principale différence réside ici, en modèle partiellement synchrone : les processus ne sont pas synchronisés entre eux, mais il existe une borne quant à la réception des messages. Cela assure la réception d'un message par un processus endéans une constante connue de tous les processus du système.

3.4 Modèles de défaillance

Les systèmes distribués sont comme des systèmes « locaux », ils ne sont pas 100% fiables. Le fait d'avoir plusieurs systèmes qui interagissent et coopèrent ensemble accroît encore plus la probabilité de subir une défaillance par rapport à un seul système isolé.

Plusieurs défaillances peuvent survenir au sein d'un système distribué et les sous-sections suivantes vont permettre de les détailler [10].

3.4.1 Défaillance définitive

La première est la défaillance définitive, aussi appelée *stopping failure*. Dans ce type de défaillance, un processus (ou nœud) stoppe tout simplement son exécution sans avertir les autres processus du système.

Le processus peut se stopper n'importe quand, au début de l'étape de l'exécution ou à la fin. Il peut aussi se stopper en plein milieu de son exécution. Dans ce cas, uniquement un sous-ensemble des messages qui devaient être envoyés a été délivré. Cela peut être n'importe quel sous-ensemble de messages, car un processus peut ne pas les produire séquentiellement.

Il est assez simple de détecter une défaillance définitive, car il s'agit d'un « crash » d'un processus qui est facilement détectable par les autres processus grâce à la mise en place de mécanismes assez simples.

3.4.2 Défaillance Byzantine

Ce type de défaillance est à ne pas confondre avec celui présenté à la section 3.4.1. Dans une défaillance Byzantine, aussi appelée *Byzantine failure*, le processus n'interrompt pas son exécution définitivement, il continue de fonctionner.

Cependant, son fonctionnement ne suit plus un comportement « normal ». Il peut générer ses messages ou son état suivant d'une façon totalement arbitraire tel que cela n'est pas prévu par la fonction de génération de messages et la fonction de transition du processus.

Ces défaillances sont plus difficiles à détecter que celles de la section 3.4.1, car le processus continue son exécution. Il faut mettre en place des mécanismes de détection de fautes assez robustes, ce qui est une des tâches les plus difficiles dans le modèle de défaillances.

3.4.3 Défaillance de communication

Dans un système distribué, les processus possèdent des canaux de communications entre eux qui leur permettent de s'échanger de l'information.

Ce type de défaillance se produit sur ces canaux. Un processus peut vouloir générer un message pour le placer sur le canal de communication, mais ce dernier peut ne pas l'enregistrer ou le délivrer correctement, ce qui revient à perdre tout simplement le message.

3.5 Problème de l'accord distribué

Le problème de l'accord est un problème fondamental du calcul distribué. En effet, les différentes tâches distribuées exécutées par les processus représentent le système distribué. Or ces processus doivent à un moment s'accorder sur la valeur finale qu'ils tentent de calculer ensemble.

C'est pour ces raisons qu'il s'agit du problème de l'accord, les processus effectuent leurs propres calculs en fonction de leur environnement et communiquent avec les autres afin d'échanger leurs données. C'est en utilisant cet échange d'information que les processus doivent s'accorder sur une certaine valeur.

Dans ce problème, il en existe plusieurs types. Détaillons les plus connus.

3.5.1 Élection d'un leader

Dans le problème de l'élection, il est nécessaire d'élire un et un seul *leader* parmi tous les processus. Le *leader* doit être élu par tous les processus du système qui doivent se mettre d'accord.

Ce cas d'accord distribué intervient dans beaucoup de systèmes distribués, car la topologie du réseau fait qu'il n'est pas totalement symétrique et un processus devient le *leader* afin d'initier l'algorithme. Une autre raison peut être une sauvegarde des ressources lorsqu'on ne veut pas que l'algorithme soit répliqué sur toutes les machines du système [8].

Cependant, la plupart des algorithmes permettant l'élection d'un *leader* ont comme hypothèse qu'un réseau en anneau est disponible afin de fonctionner correctement. Grâce au réseau en anneau, chaque processus possède un voisin de gauche et de droite et il est possible d'élire un *leader* par simple échange de messages.

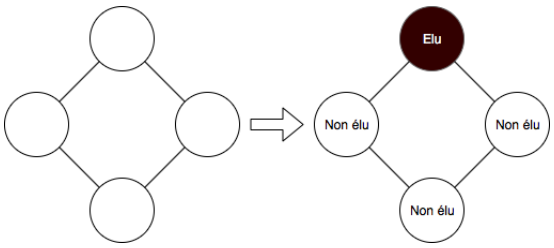

FIGURE 3.6 – Élection d'un *leader* dans un ring

3.5.2 Consensus

Le problème du consensus est un problème fondamental de l'algorithmique distribuée.

Dans les systèmes distribués, chaque processus effectue ses propres calculs localement. Ensuite, en utilisant les moyens de communication du réseau, les processus échangent leurs données calculées.
Cet échange permet de trouver une valeur commune entre tous les processus qui sera la valeur « finale » calculée par l'algorithme.

Dans le cas où aucune défaillance ne se produit, l'accord se trouve par simple échange de messages.
Cependant, le système peut subir des défaillances. Il est donc nécessaire de le rendre robuste et résistant aux différentes pannes qu'il pourrait subir.

Certains algorithmes ont une hypothèse moins forte en ne nécessitant qu'un certain pourcentage de processus s'accordent au lieu de l'entièreté, ce qui rend leurs implémentations plus simples. L'accord approximatif en est un bon exemple [10].

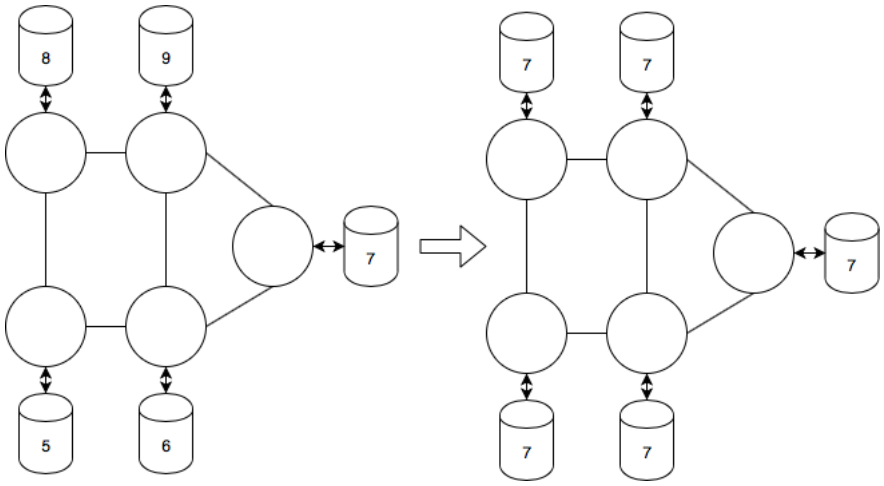

FIGURE 3.7 – Consensus sur une valeur par des processus

De manière plus formelle, le problème du consensus doit répondre à ces différents critères [10] :

• **Terminaison :** Chaque processus doit décider d'une valeur.

- **Intégrité :** La valeur décidée par l'ensemble des processus doit avoir été proposée par un des processus.
- **Accord :** Tous les processus s'accordent sur la même valeur.

3.6 Exemple

Il existe plusieurs problèmes qui nécessitent de trouver un consensus au sein de leur système distribué. Nous pouvons citer par exemple, le problème de l'accord à plusieurs valeurs (*k-agreement problem*), le problème de l'accord approximatif (*approximate agreement problem*) ou encore le problème de validation dans des bases des données distribuées (*distributed database commit problem*).

Le problème abordé dans cette section est celui concernant les bases de données distribuées qui est un problème bien connu dans le domaine de l'informatique.

Dans ce problème, une collection de processus participent au traitement d'une transaction d'une base de données. Après ce traitement, chaque processus à une opinion initiale sur la validation (*commit*) ou l'annulation (*abort*) de la transaction.
Un processus va favoriser la validation si tous ses calculs locaux avant le traitement de cette transaction se sont bien passés. Sinon, il favorise l'annulation [10].

3.6.1 Problématique

La version considérée dans ce cas est une version simplifiée, mais qui aborde le même problème. Les seuls types de défaillances considérés ici sont les défaillances définitives et pas les défaillances de communication. Pour pallier à ces dernières, il faut ajouter d'autres éléments comme la retransmission de messages pour lutter contre leur perte.

Considérons des processus communiquant par échange de messages. La topologie du réseau représente un graphe complet et le domaine d'entrée est l'ensemble des valeurs $\{0, 1\}$ où 0 représente annuler (*abort*) et 1 valider (*commit*).

En reprenant les critères du problème du consensus de la section 3.5.2, nous pouvons les adapter pour notre problème [10] :

Accord : Aucun processus ne décide de valeurs différentes.

Validité : Si un processus commence avec la valeur 0, alors 0 est la seule valeur de décision possible.
Si tous les processus commencent avec 1 et qu'il n'y a pas de défaillance, alors 1 est la seule valeur de décision possible.

Terminaison : La condition de terminaison faible dit que s'il n'y a pas de faute, tous les processus peuvent éventuellement décider. La condition forte dit que tous les processus non fautifs peuvent éventuellement décider.

3.6.2 Solution

Il existe un algorithme qui permet de résoudre ce problème : le *two-phase commit* [10].

L'algorithme suppose un processus distinct, admettons le processus 1.

Tour 1 : Tous les processus sauf le processus 1 envoient leurs valeurs initiales au processus 1 et tout processus dont la valeur initiale est 0 décide 0. Le processus 1 collecte toutes les valeurs plus sa valeur initiale dans un vecteur. Si, pour chaque position du vecteur, la valeur est 1, le processus 1 décide « 1 ». Sinon, si certaines positions contiennent 0 ou ne contiennent rien (défaillance du processus), le processus 1 décide « 0 ».

Tour 2 : Le processus 1 diffuse la valeur à tous les autres processus. Tous ces processus, excepté le processus 1, qui reçoivent un message au tour 2 et qui n'ont pas encore décidé au tour 1, décident en fonction de la valeur reçue par le processus 1.

Cet algorithme ne satisfait que la condition de terminaison faible et il est bloquant. Il existe cependant un « embellissement » du *two-phase commit* : le *three-phase commit* [10].

Tour 1 : Tous les processus sauf 1 envoient leurs valeurs initiales au processus 1 et tout processus dont la valeur initiale est 0 décide « 0 ». Le processus 1 collecte toutes les valeurs plus sa valeur initiale dans un vecteur. Si, pour chaque position du vecteur, la valeur est 1, le processus 1 ne décide pas encore, mais devient *ready*. Sinon, si certaines positions contiennent 0 ou ne contiennent rien (défaillance du processus), le processus 1 décide « 0 ».

Tour 2 : Si le processus 1 a décidé « 0 », il diffuse *decide(0)*. Sinon, il diffuse *ready*. Chaque processus recevant *decide(0)* décide « 0 » et chaque

processus recevant *ready* devient *ready*. Le processus 1 décide « 1 » s'il n'a pas encore décidé.

Tour 3 : Si le processus 1 a décidé « 1 », il diffuse *decide(1)* et chaque processus recevant *decide(1)* décide « 1 » également.

Cet algorithme satisfait la condition de terminaison forte alors que le précédent ne satisfaisait que la faible. De plus, contrairement au précédent, il a la propriété de ne pas être bloquant.

Chapitre 4

Implémentation des solutions

4.1 Contexte

Ce chapitre aborde l'implémentation des différentes solutions permettant la gestion des conflits lorsque plusieurs moteurs proactifs développés par l'Université du Luxembourg sont mis en réseau afin de gérer tous ensemble un environnement bien défini.

Les implémentations existantes permettent l'exécution d'un moteur proactif dans un environnement particulier. Ce dernier est connecté à un seul système qu'il contrôle et il exécute les tâches pour lesquelles il est programmé sur ce même système.

Cependant, dans certains cas, il est nécessaire de surveiller plusieurs systèmes ne se situant pas au même endroit, de faire des calculs locaux sur ceux-ci et de communiquer le résultat de ces calculs entre moteurs afin de prendre une décision globale sur un des systèmes ou tout simplement de façon externe.

Cette nouvelle vision des choses implique une mise en réseau de plusieurs moteurs proactifs afin qu'ils surveillent chacun un système ou un environnement spécifique de manière locale, qu'ils se communiquent les informations et qu'ils prennent ensemble les décisions qui conviennent le mieux pour conserver l'intégrité de l'environnement global.

4.1.1 Problématique

Cette mise en réseau de plusieurs moteurs proactifs engendre plusieurs problèmes qui n'existaient pas auparavant lors de l'utilisation d'un seul et unique moteur.

Cette problématique se focalise sur deux axes essentiellement.

Le premier axe est la communication entre les différents moteurs proactifs. En ajoutant une dimension « réseau » au système, il faut nécessairement un moyen de communication au sein de ce système.

Chaque moteur va s'exécuter dans son environnement local, récolter des données relatives à celui-ci et effectuer des calculs. Une fois ces opérations terminées, chaque moteur va devoir communiquer le résultat des calculs et éventuellement son état actuel. En se basant sur les méthodes de communications de la section 3.2, une d'elles va devoir être retenue et être implémentée.

Le deuxième axe découle du premier détaillé dans le paragraphe au-dessus, il s'agit de la gestion de conflits. Grâce à la mise en place d'une méthode de communication, chaque moteur sera capable de connaître l'état actuel des autres moteurs proactifs du réseau. Ceci fait survenir un nouveau problème, car chaque moteur effectue ses calculs locaux et communique les résultats au réseau. Cependant, le système n'est pas à l'abri de conflits entre décisions locales de différents moteurs (décisions contradictoires par exemple).

Chaque moteur, en plus de ses décisions locales, va devoir intégrer une logique de prise de décision répartie (*conflict handling*). Cette logique va intervenir sur plusieurs éléments essentiels de la gestion de conflits (détection et résolution de conflits).

En intégrant ceci, les moteurs seront capables de prendre des décisions à l'échelle locale en utilisant leurs données récoltées, mais également à l'échelle globale en utilisant les données et l'état des autres moteurs du système.

4.1.2 Critères d'évaluation

L'implémentation de ce type de solution doit répondre à plusieurs critères permettant d'évaluer l'implémentation d'une solution répondant à la problématique de la section 4.1.1.

Les critères énoncés ci-dessous seront utiles afin d'évaluer la solution mise en place aux sections 4.2 et 4.3. Chaque critère permet d'évaluer les choix effectués parmi ceux qui étaient disponibles pour élaborer la solution.

Voici quelques critères permettant de respecter au maximum l'architecture actuelle du moteur proactif :

1. Le moyen de communication implémenté doit être « léger », il ne doit pas ralentir l'exécution du moteur.
2. Le moyen de communication doit transmettre le moins de messages possible afin de limiter les pertes et les délais de transmission.
3. Le système doit prendre ses décisions de manière décentralisée.
4. En rejoignant le critère précédent, aucun processus du système ne doit être supérieur aux autres.
5. Le système doit être exempt de toute situation d'interblocage.
6. Le système doit être robuste aux défaillances.

4.1.3 Questions de recherche

Ces différentes problématiques permettent d'énoncer quelques questions de recherche sur lesquelles la suite de ce document va se pencher.

Premièrement, quels sont les scénarios locaux à développer pour chaque moteur ? Avant de communiquer avec les autres et prendre des décisions, un moteur doit utiliser les données qu'il recueille localement pour agir également de manière locale. Cette première question découle directement du découplage des moteurs à différents endroits afin de les positionner à des endroits stratégiques.

Ensuite, étant donné la multitude de moteurs proactifs travaillant en parallèle, quel moyen de communication préférer parmi ceux définis dans la section 3 sur les algorithmes distribués ? Dus au découplage des moteurs, ils ne se situent plus à proximité les uns des autres. Il est donc nécessaire d'ajouter une dimension de communication fiable et légère permettant l'échange de données nécessaires à la gestion de conflits.

Dernièrement, concernant les potentiels conflits dans l'exécution des différents moteurs, quelle politique de management des moteurs mettre en œuvre afin de détecter, résoudre et prévenir les conflits potentiels pouvant survenir ? Chaque moteur est indépendant quant aux données, mais aussi quant aux décisions prises localement. Pour gérer les conflits potentiels entre les différentes exécutions, il est nécessaire d'ajouter une dimension de logique répartie entre les moteurs par le biais d'un algorithme de prise de décision répartie.

4.2 Moyen de communication

4.2.1 Introduction

La mise en réseau de plusieurs moteurs nécessitant des capacités de communication, il est nécessaire de se concentrer sur cette partie en premier lieu.

Comme vu dans la partie sur les algorithmes distribués au chapitre 3, il existe plusieurs méthodes de communication pour un système distribué : l'envoi de messages entre processus et la communication grâce à une mémoire partagée (section 3.2).

La méthode qui a été choisie est l'implémentation d'une mémoire partagée commune à tous les moteurs proactifs du système. Cette dernière est modélisée à la figure 4.1.

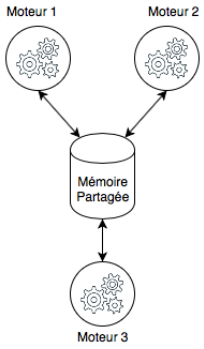

FIGURE 4.1 – Mémoire partagée entre les moteurs

En opposition à la mémoire partagée, l'envoi de message est une tâche assez lourde. Il est nécessaire d'implémenter la transmission (envoi, réception, perte de messages, etc.), une file d'attente pour gérer la différence entre l'afflux et le traitement, mais également la réplication des données sur chaque moteur qu'il est nécessaire d'actualiser en fonction des messages reçus.

Cependant, la mise en place d'une mémoire partagée nécessite l'implémentation d'éléments spécifiques comme un mécanisme de verrou en lecture et en écriture, une API d'accès à distance, un rafraîchissement constant des données, etc.

La décision du choix d'une mémoire partagée comme moyen de communication réside dans le nombre de modifications à fournir. Pour l'échange de messages, l'exécution du moteur proactif devait être revue afin d'y intégrer

un envoi et une réception de messages. Ensuite, il faut également s'assurer de la bonne réception d'un message par tout le monde, un mécanisme de retransmission en cas de perte est inévitable. De plus, l'exécution se faisant à des vitesses différentes, chaque moteur nécessite un *buffer* afin de stocker le surplus de messages.

Toutes ces raisons ont fait que la méthode de communication choisie a été la mémoire partagée grâce à sa plus grande adaptabilité au système.

4.2.2 Architecture

La communication entre les moteurs proactifs prenant une forme de mémoire partagée, il faut désormais modéliser son architecture. Il a fallu choisir entre deux formes différentes de mémoire partagée, une virtuelle (appelée également simulée) et une physique (appelée également réelle).

Le choix effectué ici est d'utiliser une mémoire physique, car elle n'est pas spécialement un composant présent dans le système, elle peut se trouver dans le *cloud* par exemple. Celle-ci est modélisée sous une forme physique grâce à une base de données.

Il s'agit ici d'une base de données MySQL à part entière entièrement distincte de celles appelées « locales » où les moteurs proactifs stockent leurs données récoltées localement. Contrairement à la mémoire partagée qui doit venir se greffer au système, ces dernières sont déjà présentes dans l'architecture du moteur proactif.

La base de données possède des tables, chacune d'elles représente un type de données provenant d'une même source, un capteur par exemple. Chaque table peut posséder plusieurs lignes représentant une source de données. Si plusieurs sources de données existent, il est possible d'ajouter plusieurs lignes et de les différencier grâce à leur identifiant.

Une convention de nommage a été définie afin d'identifier chaque table partagée et pouvant être accédée pour tous les moteurs. Elles commencent par la mention *SM_[XXX]* dans le nom de la table. Cela permet de reconnaître les tables appartenant à la mémoire partagée (*shared memory* ou SM).

Chaque table possède plusieurs attributs en fonction des choses que la source doit stocker. Chaque table possède au moins un identifiant et une valeur. L'identifiant est un *integer* et la valeur varie en fonction de l'élément récolté.

Prenons comme exemple un capteur récoltant la température extérieure. Chaque entrée dans la table doit fournir un identifiant, permettant d'identifier de manière unique le capteur, une valeur de type *float*, représentant la température extérieure, mais également un *timestamp* qui indique le moment de la dernière actualisation de la valeur.

Il est peut-être intéressant d'ajouter un *timestamp* aux sources de données, cela permet de déceler une anomalie dans la mise à jour de la donnée. Si la source s'est arrêtée de fonctionner et que sa valeur est trop ancienne, il sera peut-être nécessaire de ne pas prendre le risque d'utiliser cette valeur qui peut être obsolète.

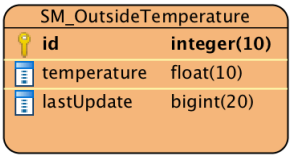

FIGURE 4.2 – Table contrôlant des capteurs de température extérieure

4.2.3 Interface d'accès

La section précédente a abordé le fait qu'il soit possible que la mémoire partagée ne soit pas présente physiquement au même endroit que les moteurs proactifs.

En effet, la gestion de la mémoire partagée peut être une tâche fastidieuse pour des utilisateurs lambda. Cette complication offre plusieurs solutions qui peuvent diminuer voire même rendre inexistante cette complexité de maintenance.

Tous les utilisateurs ne sont pas spécialement des experts en informatique et le fait de laisser la mémoire partagée au sein de leur maison pose donc plusieurs problèmes qui vont être abordés.

La solution retenue ici et ayant le plus de sens est l'utilisation d'une interface d'accès permettant de « communiquer » avec la mémoire partagée.

Un des principaux points forts de cette solution est la maintenance du logiciel, chaque appel à la mémoire partagée est centralisé au même endroit et permet de centraliser la maintenance.
Cette centralisation des appels permet une mobilité de mémoire partagée

comme détaillée à la section 4.2.2. La mémoire peut être déplacée sans devoir modifier chaque appel dans le code comme tout est centralisé au même endroit.

Un autre point fort, qui n'est cependant pas présent dans tous les systèmes, est la sécurité. Si la mémoire partagée contient des données sensibles, il est nécessaire d'assurer la sécurité de ces données.
Pour éviter toute fuite de données potentielle, il est également possible d'externaliser la mémoire partagée afin de confier la confidentialité et l'intégrité des données à un tiers qui doit être un expert dans ce domaine.
Il est important de citer que l'externalisation permet aussi d'assurer une disponibilité plus importante en cas de défaillance quelconque. La mémoire partagée étant le point de communication central entre moteurs, sa disponibilité est un élément critique.

Il est important de préciser que le point fort du paragraphe précédent regroupe les trois objectifs qui sont : *Disponibilité - Intégrité - Confidentialité* dans la sécurité des systèmes d'information. La norme traitant cette sécurité est la norme ISO/CEI2700 [3].

Application Programming Interface

La mémoire partagée pouvant être externalisée, il est impératif d'avoir une interface rendant transparents les appels à cette mémoire pour le moteur proactif. Au moyen d'une *application programming interface* (API), les moteurs proactifs peuvent avoir accès aux données partagées sans se soucier de savoir où se trouvent celles-ci.

L'architecture du moteur proactif possède déjà un « wrapper » qui lui permet de se connecter à une base de données distante ; celui-ci va être réutilisé.

L'implémentation de ce « wrapper » se base sur un principe assez connu issu des patrons de conception (design patterns) : le *proxy*. Grâce à ce pattern, il est possible pour une classe de représenter les fonctionnalités définies par une autre classe. Il est également possible que la classe qui utilise l'autre change le comportement ou l'implémentation de celle-ci à des fins d'optimisation ou autre.

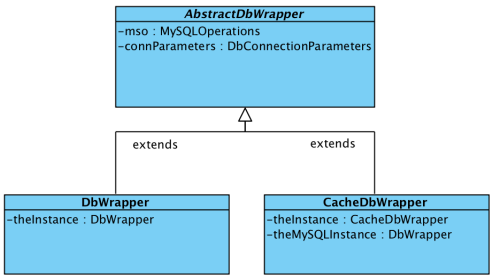

FIGURE 4.3 – Proxy utilisé par le wrapper du moteur proactif

La classe *AbstractDbWrapper* est une classe abstraite qui remplace l'interface dans le schéma UML de base du *proxy*, car elle contient des attributs nécessaires à la connexion et à l'accès de la base de données.

La classe *DbWrapper* étend la classe abstraite. C'est au sein de cette dernière que sont implémentées toutes les requêtes SQL vers la mémoire partagée.

La classe *CacheDbWrapper* est le *proxy*. Elle implémente également la classe abstraite, mais elle effectue tous ses appels grâce à l'instance de *DbWrapper* qu'elle possède. Ce procédé permet au *proxy* d'implémenter un cache par-dessus les requêtes pour réduire les appels à la mémoire partagée si certaines données peuvent être mises en cache.

La figure 4.4 représente les appels entre ces différentes classes sous forme de composants logiques.

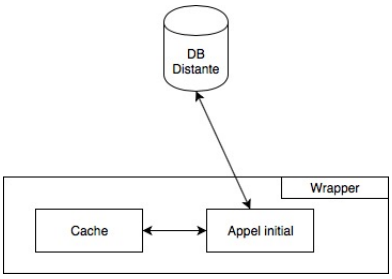

FIGURE 4.4 – Wrapper du moteur proactif

Ce « wrapper » est composé d'un cache (le *proxy*) et d'appels initiaux et constitue notre API. Cette dernière permet à un moteur proactif de se connecter à la mémoire partagée sans en connaître les détails de l'implémentation, il utilise uniquement les méthodes mises à sa disposition.

4.2.4 Verrous

La mémoire partagée étant accédée par différents moteurs distincts, il est nécessaire de préserver l'intégrité des données au sein de celles-ci.

En effet, il existe plusieurs problèmes quand on travaille avec des processus concurrents qui essaient de lire ou modifier une donnée à laquelle chacun a accès. Dans notre cas, l'accès n'est pas restreint aux moteurs, ils peuvent lire et écrire sur la mémoire partagée sans contrainte particulière.

Cependant, il est important d'avoir un mécanisme de verrouillage permettant de prévenir cette concurrence d'accès entre moteurs proactifs.

Tout d'abord, la mémoire partagée a besoin d'un verrou en écriture. Ce premier verrou est trivial, car il n'est pas possible pour deux moteurs souhaitant modifier une même donnée simultanément de le faire, ils doivent s'ordonnancer.
Un moteur doit obtenir un verrou qui lui permet de modifier la donnée. Une fois la modification effectuée, il relâche ce verrou qu'un autre moteur peut prendre pour effectuer ses modifications.

Mais l'écriture n'est pas le seul cas qui pose problème, la lecture peut aussi l'être. Ce cas ressemble au problème de la réservation de la dernière place à bord d'un avion. Deux personnes souhaitent réserver la dernière place d'un vol. Ils accèdent tous les deux à la plateforme et voient qu'il ne reste qu'une seule place disponible. Ils décident donc de l'acheter et ils effectuent cette action en même temps. Comme aucun mécanisme n'est mis en place, les deux voyageurs ont réservé la même place.

Dans des cas limites du système, il est possible de reproduire les problèmes énoncés auparavant. Si on prend une donnée partagée, admettons que celle-ci puisse être modifiée par deux moteurs proactifs différents et agissant comme suit :
« Lorsque je veux décrémenter la valeur, si elle est supérieure ou égale à 1 j'exécute l'instruction, s'il est strictement inférieur à 1, je ne l'exécute pas, car je ne peux pas descendre sous 0. »
Admettons que ces deux moteurs effectuent cet exemple en même temps, ils vérifient la valeur qui s'avère être 1. Ils décident alors de la décrémenter. L'intégrité de la donnée est mise à mal, car désormais la valeur vaut -1. Cela peut mettre en péril tout le système si rien n'est prévu pour gérer ce type de cas limite.

Un mécanisme de verrou natif existe en MySQL, il s'agit de verrous associés aux connexions à la base de données. Il en existe de deux types :

1. Un verrou de lecture (*read*). Il empêche aux autres connexions toute modification des données verrouillées. Cependant, il permet toujours la lecture de celles-ci (lecture seule).
2. Un verrou d'écriture (*write*). Il empêche les autres connexions de lire ou modifier les données verrouillées (aucun accès).

Le choix le plus judicieux est d'implémenter le verrou en écriture pour éviter tout problème pouvant affecter l'intégrité des données.

```
1 LOCK TABLES SM_People WRITE;
2
3 UPDATE SM_People
4 SET nbr = nbr - 1, lastUpdate = now()
5 WHERE id = 1 AND nbr > 0;
6
7 UNLOCK TABLES;
```

Listing 4.1 – Exemple de verrou *write* en mémoire partagée

Ensuite, en plus d'avoir un verrou, il peut être intéressant de travailler avec des transactions dans certains cas.

Certaines actions peuvent nécessiter plusieurs requêtes sur la mémoire partagée qui ne peuvent pas être dissociées. Si une des requêtes de la transaction échoue, il faut annuler l'ensemble des opérations déjà effectuées et réessayer plus tard.

4.2.5 Mise à jour

La mémoire partagée contient les données que les différentes sources d'information récoltent. Il est impératif pour celle-ci de maintenir ses données à jour constamment pour que les moteurs proactifs qui les utilisent se basent sur des éléments fiables.

Pour maintenir une mémoire partagée la plus complète possible avec des données constamment à jour, plusieurs solutions ont été envisagées avec chacune leurs avantages et leurs inconvénients.

La première solution envisagée est celle d'un moteur proactif contrôlant la mémoire partagée. Un moteur peut être « attaché » à celle-ci et effectue

des appels aux autres moteurs proactifs afin d'obtenir une mise à jour des données si cela est nécessaire. Il s'agit d'un *pull* des données vers la mémoire (figure 4.5).
Cependant, la section 4.2.3 décrit la mémoire partagée comme une mémoire pouvant être externalisée, une telle solution est donc difficilement envisageable, car si nous n'avons pas le contrôle sur la base de données partagée, il est impossible d'y « attacher » un moteur.

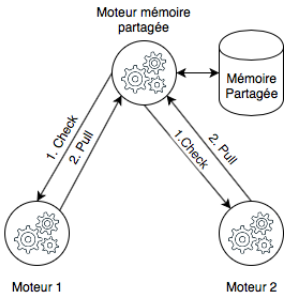

FIGURE 4.5 – Pull des données vers la mémoire partagée

L'autre solution abordée est la mise à jour directement par les moteurs proactifs. Chacun d'entre eux récolte et utilise des données localement, mais en plus de cela, chaque moteur est responsable de la mise à jour de celles-ci en mémoire partagée. Il s'agit d'un *push* des données vers la mémoire (figure 4.6).

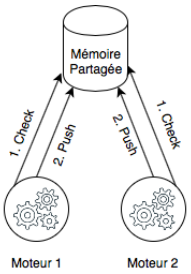

FIGURE 4.6 – Push des données vers la mémoire partagée

Chaque moteur proactif ayant déjà ses règles à exécuter, il va falloir trouver un moyen peu gourmand et assez rapide d'effectuer la mise à jour tout en évitant de bloquer l'exécution des scénarios définis.

Pour respecter ces conditions, la solution retenue est la deuxième (modèle push des données). En utilisant ce fonctionnement, il n'est pas nécessaire de

modifier le fonctionnement général du moteur ni d'ajouter un moteur supplémentaire. Il est possible de déléguer la responsabilité de la mise à jour de la mémoire partagée à une règle spécifique.

Cette règle devra être la plus petite possible pour ne pas ralentir l'exécution des autres règles. Pour rendre cela possible, chaque source (un capteur par exemple) voyant ses données partagées aura une règle s'occupant d'elle.

Pour continuer, reprenons le même exemple détaillé dans la section 4.2.2 portant sur un capteur de température extérieure.

Une règle désirant mettre à jour la mémoire partagée est implémentée en utilisant l'interface commune à toutes les règles définies au sein d'un moteur. Elle respecte exactement la même architecture et implémente les cinq méthodes « de base » de toute règle (section 2.3.1.3).

La méthode *dataAcquisition()* récupère le « wrapper » et les données du capteur.

Afin de ne pas ralentir l'exécution de la règle, il est possible de vérifier si la valeur locale diffère de celle en mémoire partagée. Une lecture supplémentaire n'influe que très peu sur le temps d'exécution, mais cette lecture peut faire économiser une écriture prenant plus de temps.
Cette vérification peut se faire dans les méthodes *activationGuards()* et *conditions()* en fonction de la taille de la condition.

La partie *actions()* permet, via le « wrapper », d'écrire la nouvelle valeur en mémoire partagée.

Pour terminer, comme la règle doit être présente à chaque itération afin d'assurer l'actualisation des données, elle se clone elle-même.

```
1 public class SM_UpdateOutsideTemperature extends
     AbstractRule {
2
3     private OutsideTemperature outsideTemperature;
4     private AbstractDbWrapper dbWrapper;
5
6     @Override
7     protected void dataAcquisition() {
8         outsideTemperature =
              getOutsideTemperatureSensor();
9         dbWrapper = getDataNativeSystem();
```

```
10      }
11
12      @Override
13      protected boolean activationGuards() {
14          return dbWrapper.getOutsideTemperature() !=
                outsideTemperature.getTemperature();
15      }
16
17      @Override
18      protected boolean conditions() {
19          return true;
20      }
21
22      @Override
23      protected void actions() {
24          dbWrapper
25              .updateOutsideTemperature(
                    outsideTemperature.getTemperature(),
                    outsideTemperature.getLastUpdate());
26      }
27
28      @Override
29      protected boolean rulesGeneration() {
30          return createRule(this);
31      }
32
33      @Override
34      public String toString() {
35          return this.getClass().toString();
36      }
37  }
```

Listing 4.2 – Règle mettant à jour le capteur de température extérieure

Une dernière chose à prendre en compte, ce sont les paramètres du moteur. Il en existe trois : F, N et P. Il faut les ajuster pour que le nombre de règles (N) soit suffisant pour accueillir les règles locales et de mise à jour, mais aussi ajuster la fréquence (F) à la baisse afin qu'une itération soit la plus courte possible pour ne pas laisser le temps aux valeurs partagées de « périmer ».

4.3 Prise de décision répartie

4.3.1 Introduction

La problématique (section 4.1.1) énonçait deux choses. Premièrement, la mise en réseau de moteurs proactifs impliquait la mise en place d'un moyen de communication, cela a été mis en place à la section 4.2.

La deuxième chose associée à la problématique découlait directement de cette communication. Les moteurs, en s'échangeant de l'information, doivent se mettre d'accord à un moment donné sur les actions à effectuer sur le système « global ».

Cet accord peut-être atteint grâce à l'échange d'informations via la mémoire partagée. Cependant, un moteur du réseau ne connaît pas l'état des autres et une décision locale peut potentiellement entrer en conflit avec l'état des autres moteurs.

Une politique de management va devoir être mise en place. Dans un premier temps, il faudra éviter les conflits autant que possible. Dans un deuxième temps, certains conflits étant inévitables, il faudra établir un algorithme de prise de décision afin de résoudre les conflits qui se présentent.

Cette section va détailler la mise en œuvre de la politique de management dans le prototype afin d'instaurer une prise de décision répartie dans le système logiciel.

Avant d'aborder la prise de décision en elle-même, il est impératif de définir succinctement un conflit.

Une situation est conflictuelle lorsque, dans notre cas, deux moteurs constatent une opposition entre l'exécution d'une action de la part d'un moteur et l'état d'un autre moteur. Ce problème s'apparente beaucoup à celui rencontré dans l'état de l'art sur les algorithmes distribués, le problème de l'accord distribué (section 3.5).

C'est ce problème que le développement au sein de cette section cherche à résoudre. Les moteurs, face à divers conflits entre eux, vont devoir trouver un accord satisfaisant tout le monde et solutionnant ces diverses situations.

4.3.2 Objectifs

La prise de décision, aussi appelée négociation entre moteurs, doit subvenir à deux objectifs majeurs :

— Le « *conflict handling* », appelé aussi gestion de conflits, qui est composée de trois points :
  1. La détection de conflits
  2. La résolution de conflits
  3. Éventuellement, la prévention de conflits
— L'optimisation de la prise de décision.

Il est clair que la détection et la prévention se recoupent assez fort même si ce n'est pas le même concept. En effet, la prévention est plus difficile à mettre en place que la détection de conflits, elle permet de déterminer à l'avance si un conflit va se produire avant même son apparition.
En sachant comment « modéliser » un conflit, il est plus facile de le détecter. Or, avec de la prévention, un conflit qui peut se produire pour la première fois au système devra être évité avant même son apparition. C'est cela qui rend la prévention plus difficile que la détection.

Ces deux objectifs sont à garder en tête tout au long du développement de la solution suivante, car c'est autour de ceux-ci qu'elle va se construire.

4.3.3 Détection des conflits

Tel que défini à la section 2.3.1.7, le moteur exécute des scénarios en fonctionnant par itérations. Les scénarios définis ne sont qu'un enchaînement de règles vérifiant une ou deux conditions tout au plus. La première règle composant le scénario est appelée le « point d'entrée ».
Cette règle vérifie généralement des données locales disponibles dans la base de données du moteur.

Pour détecter les conflits avant que ceux-ci ne se produisent, il est important de posséder un moyen de communication puisque c'est grâce à celui-ci qu'il est possible de s'échanger de l'information afin d'atteindre cet objectif.

Admettons un scénario qui est composé de plusieurs règles dont un point d'entrée est plusieurs actions possibles. En le définissant, il est possible d'apercevoir des conflits qui peuvent survenir avec l'état d'autres moteurs.
Par exemple, si le moteur effectue son action A, mais que le moteur se trouve à cet instant dans un état X, cela provoque un conflit. Le moteur voulant effectuer son action doit donc détecter qu'un conflit va survenir en exécutant A et en le détectant, il peut adapter son scénario en patientant ou en exécutant une action B.

Pour pallier ces problèmes, la détection de conflits est utilisée. Lors de la configuration des scénarios, plusieurs règles peuvent être intercalées entre la règle appelée « point d'entrée » et la règle finale contenant une action.

Ces règles ont un objectif fondamental. En utilisant la mémoire partagée mise en place à des fins de communication entre moteurs, un scénario peut adapter son comportement général s'il détecte un conflit afin de l'éviter.
Les données en mémoire partagée sont actualisées à chaque itération permettant de limiter les données « périmées ».

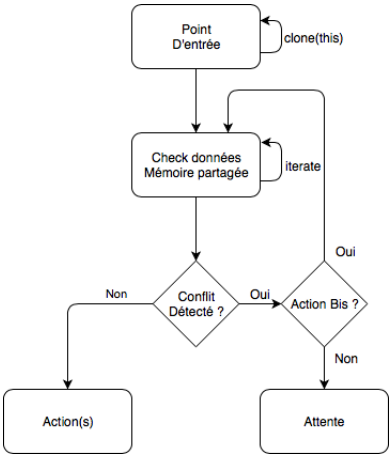

FIGURE 4.7 – Schématisation de la détection de conflits

La figure 4.7 schématise les propos tenus juste avant.

Un scénario utilisant de la détection de conflit possède une règle « point d'entrée » qui est la règle de départ.

Une fois que la condition de base est vérifiée par la règle de départ, le scénario est lancé. C'est à cet endroit que sont placées les règles de détection. Le scénario possède une ou plusieurs règles qui consistent à récupérer des données de la mémoire partagée et vérifier si cela viole les conditions prévues dans celles-ci (il peut avoir plusieurs règles, d'où le *iterate*).

Après ou pendant la vérification de ces conditions, deux cas sont possibles et sont traités de la même manière que ce soit après ou durant la vérification.
Si un conflit est détecté, le scénario peut prévoir plusieurs choses. Soit il vérifie d'autres conditions et si elles ne détectent pas de conflit, une action *bis* est exécutée (une action peut aussi être non-conflictuelle et être exécutée

sans nouvelle vérification), soit il ne prévoit rien et il se contente d'attendre une itération ultérieure afin de vérifier une nouvelle fois ces conditions.
S'il n'y a pas de conflit, le scénario se termine normalement et l'action prévue initialement est exécutée.

Bien sûr, la vérification sur la mémoire partagée ne se fait pas que sur les données en elles-mêmes. Les sections 4.2.2 et 4.2.5 avaient détaillé la possibilité d'ajouter un *timestamp* pour chaque source de données. Ce dernier permettait de vérifier la dernière actualisation de la donnée.
Avant d'utiliser la donnée, les règles de détection de conflits peuvent donc s'assurer que celle-ci n'est pas « périmée ». Si c'est le cas, elles peuvent interpréter cela comme un conflit pour ne pas prendre de risque.

Une dernière chose, qui ne fait pas partie de la détection de conflits, mais qui en découle, est le « passage d'ordres ».

La mémoire partagée est une solution qui permet aux processus (ici des moteurs proactifs) de communiquer ou de manière générale, de s'échanger de l'information. Grâce à ceci, il est possible pour un moteur d'envoyer un « ordre », à travers la mémoire partagée, à un autre moteur. Cet « ordre » permet à un moteur particulier de piloter des éléments qui ne sont pas sous son contrôle direct (il peut s'agir d'actionneurs par exemple).

En reprenant la figure 4.7, il est possible que l'action *bis* soit une action sur un élément d'un autre moteur. Le moteur détectant un conflit et qui ne possède pas d'action alternative sous son contrôle peut utiliser ce passage d'ordres pour demander l'annulation de l'action en cours d'exécution par un autre moteur par exemple.
Ce concept permet d'étendre le champ des possibilités d'un moteur : en plus de piloter le système qu'il contrôle, il peut passer des ordres pour piloter des éléments d'un autre système (attaché à un autre moteur) pour autant que ces éléments soient pris en compte dans la configuration de la mémoire partagée.

Une modélisation possible est celle représentée à la figure 4.8, cela représente un élément d'un système qui peut être un actionneur, un capteur ou autre. La table de l'élément partagé pouvant recevoir un « ordre » possède une ou plusieurs colonnes modélisant les « ordres » par des booléens.
La colonne *state* est arbitraire, elle représente l'état pour un actionneur, mais elle peut très bien se transformer en *value* ou autre chose.

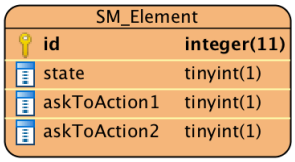

FIGURE 4.8 – Modélisation d'ordres dans la mémoire partagée

Le terme « passage d'ordre » est mis entre guillemets, car il s'agit plus d'un désir/souhait que d'un ordre en tant que tel. Malgré la détection de conflits, il se peut qu'un « ordre » soit posé, mais qu'il ne soit pas considéré par l'autre moteur, car il entre en conflit avec son environnement sans avoir été détecté au préalable par le moteur initial.

Pour résumer la partie sur la détection de conflits qui a été construite, plusieurs choses peuvent être mises en place en amont pour permettre de les éviter.

— Il est possible de mettre en place des règles supplémentaires faisant intervenir la mémoire partagée afin de détecter des conflits potentiels.
— En détectant un conflit grâce aux données de la mémoire partagée, un moteur peut prévoir dans son scénario une action secondaire (ou action *bis*) permettant de dévier son exécution vers une situation non conflictuelle.
— La mémoire partagée peut aussi être utilisée afin de passer des « ordres » d'un moteur à un autre permettant de piloter des éléments partagés.

4.3.4 Résolution des conflits

La section 4.3.3 détaille les ajouts possibles afin de détecter des conflits pouvant se produire et les moyens mis en place afin de les éviter.
Cependant, il n'a pas toujours possible de détecter (ou d'éviter) un conflit. Il existe des situations qui ne peuvent être prévues ou encore des cas où il n'est pas possible de solutionner un conflit.

Pour pallier ces conflits résiduels qui ne peuvent être évités et qui peuvent se produire malgré une probabilité d'apparition inférieure à un système sans détection de conflits, il est impératif d'ajouter une politique de management des conflits appelée également prise de décision répartie entre les moteurs.

Les conflits pouvant se produire au sein du système distribué proviennent des ordres passés en mémoire partagée et détaillés à la section 4.3.3. En effet,

un moteur voulant effectuer un ordre l'écrit en mémoire partagée pour un élément qu'il ne possède pas. Ne possédant pas cet élément, il ne possède pas non plus le scénario associé à celui-ci. Sans scénario, il est impossible pour le moteur de savoir quelle condition vérifier afin de détecter et éviter un conflit.

En ajoutant des ordres, il faut désormais ajouter une dimension de résolution des conflits pour garder une cohérence dans l'exécution globale de nos scénarios.

La résolution des conflits s'apparente beaucoup au problème de l'accord distribuée à la section 3.5. Face à un conflit, il va falloir faire intervenir l'ensemble du système pour le solutionner. Ce problème fondamental de l'algorithmique distribué peut être traité de plusieurs manières dont deux étant les plus connues : l'élection d'un *leader* ou l'atteinte d'un consensus.

L'orientation choisie pour la résolution de conflits est l'atteinte d'un consensus par l'ensemble des processus du système. La raison de ce choix est purement arbitraire. Cependant, un critère voulu est de ne pas avoir de processus « supérieur » aux autres et c'est pour cette raison que l'élection d'un *leader* n'a pas été choisie.

Cette résolution va se faire par un algorithme de votes dans lequel chaque moteur interviendra afin d'avoir un avis représentatif de l'ensemble du système.

Comme l'exemple de la section 3.6, l'algorithme est basé sur le vote. Un moteur ayant besoin de l'avis d'autres moteurs avant de résoudre un conflit, il le fait savoir et chaque moteur lui répond afin de valider ou non la demande initiale.
Voici l'algorithme détaillé de manière informelle.

1. Un moteur souhaite effectuer une action : il le communique aux autres.
   - Chaque moteur vérifie si l'action est en conflit avec son état global.
     (a) Si oui, il invalide la décision (KO).
     (b) Si non, il valide la décision (OK).
2. Le moteur récupère les décisions. 2 options s'offrent à lui :
   (a) Il effectue son action prévue.
   (b) Il patiente ou bascule vers une autre action (action bis).

Avant de détailler la construction de l'architecture de l'algorithme et son

fonctionnement, il est impératif de se pencher sur le mode communication des moteurs.

On peut apercevoir que l'initiation d'une décision passe par une communication entre moteurs. La communication entre moteurs utilisant la mémoire partagée, elle va être utilisée pour échanger les informations concernant les décisions et les votes des moteurs.

Premièrement, le moteur initie une décision de résolution de situation conflictuelle dans la mémoire partagée. Pour faire ceci, il existe une table qui regroupe toutes les décisions initiées.

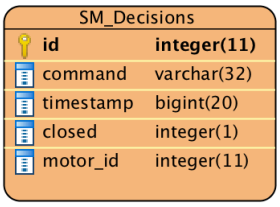

FIGURE 4.9 – Modélisation des décisions dans la mémoire partagée

Comme on peut le constater à la figure 4.9, cette table permet de regrouper toutes les décisions existantes et de garder un historique complet de ces dernières. Chaque décision possède un identifiant, un string *command*, un *timestamp*, un booléen *closed* et un identifiant de moteur.

Le string *command* indique la décision en question, ce sont généralement des constantes connues de tous les moteurs. Par exemple, cela peut être l'exécution d'un actionneur. Le *timestamp* indique l'heure d'initiation de la décision, le booléen *closed* indique si la décision est prise ou non (décision terminée) et l'identifiant de moteur permet d'identifier le moteur ayant écrit cette décision en mémoire partagée.

Maintenant que la mémoire partagée peut accueillir une décision, il faut également qu'elle puisse accueillir des votes associés à celle-ci. Il faut commencer par aborder la relation entre les votes et les décisions ainsi que leur stockage. Chaque décision possède de 0 à plusieurs votes, il s'agit donc d'une relation *0-N* entre *SM_ Decisions* et *SM_ Votes*.

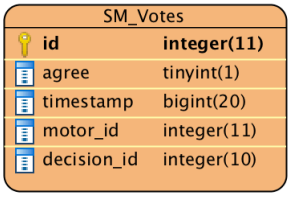

FIGURE 4.10 – Modélisation des votes dans la mémoire partagée

Comme montré à la figure 4.10, une table permet de regrouper tous les votes qui ont eu lieu. Cela permet de conserver un historique complet des votes de chaque moteur dans le temps. Chaque vote possède un identifiant, un booléen *agree*, un *timestamp*, un identifiant de moteur et un identifiant de décision.

L'identifiant de moteur permet de savoir de quel moteur provient le vote et le *timestamp* permet de connaître l'heure exacte du vote. Le booléen *agree* contient *true* si le moteur identifié par *motor_id* n'est pas opposé à la décision, autrement dit la décision ne pose pas de conflit avec l'état du moteur qui vote. Si c'est le cas, alors *agree* contient *false* pour indiquer qu'il y a un conflit potentiel.

Après avoir détaillé le stockage des décisions et des votes en mémoire partagée, l'algorithme de vote et la prise de décision finale vont être détaillés.

En reprenant l'algorithme informel détaillé au début de cette section, détaillons pas à pas son exécution au sein du moteur grâce une mise en situation fictive.

Un scénario impliquant de la détection de conflits se déclenche. Plusieurs conditions sont vérifiées et l'action ne peut être exécutée, car cela provoque une situation conflictuelle. Le scénario prévoit une action alternative qui utilise un élément ne se trouvant pas sous son contrôle. En utilisant le « passage d'ordres » de la section 4.3.3, il indique au moteur possédant cet élément qu'il souhaiterait l'actionner.

Le moteur voit l'ordre écrit en mémoire partagée, mais il n'est pas certain que son exécution ne va pas poser problème puisque cette action ne fait pas partie de ses scénarios locaux. Il initie alors une décision. Pour effectuer ceci, il utilise l'API mise en place afin d'accéder à la mémoire partagée (section 4.2.3). Il indique l'ordre qu'il aimerait faire, son identifiant de moteur, l'heure actuelle et met la colonne *closed* à *false*.

Ensuite, c'est à chaque moteur de voter pour ou contre cette décision. Le moteur fonctionnant sur base d'exécution d'un ensemble de règles à chaque itération, chaque moteur possède deux règles qui s'exécutent à chaque itération. Détaillons ces règles.

La première est la règle permettant la réponse par un vote aux décisions. C'est cette règle qui contient la « connaissance » en matière de résolution de conflits. Dans un premier temps, la règle s'occupe de récupérer toutes les décisions pour lesquelles le booléen *closed* est mis à *false*.

Pour chaque décision récupérée, le moteur effectue ceci :

- Si le moteur a déjà ajouté son vote à cette décision, il passe à la suivante ou se termine si c'était la dernière.
- Sinon, deux cas se présentent :
  - La décision émane du même moteur qui exécute la règle de vote. La règle ajoute un vote avec *agree* à *true*.
  - Si la décision émane d'un autre moteur. La règle possède différentes possibilités parmi lesquelles elle peut choisir en fonction de la décision. Chaque possibilité est une suite de conditions logiques permettant de détecter s'il y a bien conflit et d'adapter le vote final en fonction. Une fois les conditions évaluées, le vote est construit avec *agree* affecté à la bonne valeur booléenne en fonction du conflit existant ou non. Ce vote est ensuite ajouté à la décision en mémoire partagée.

Pour mieux illustrer la conception d'un tel type de règle au sein d'un moteur, elle a été résumée en pseudo-code Java ci-dessous.

```
1  class HandleVotes {
2
3      void dataAcquisition() {
4          sharedMemory = getSharedMemoryWrapper();
5          states = getElementsStates();
6      }
7
8      boolean activationGuards() {
9          return true;
10     }
11
12     boolean conditions() {
13         return true;
```

```
14      }
15
16      void actions() {
17          decisions = getAllDecisionNotClosed();
18
19          for (decision : decisions){
20              if (sharedMemory.isThereAVote(decision.
                    getId, MOTOR_ID)){
21                  continue;
22              }
23
24              if (decision.getMotorId == MOTOR_ID) {
25                  sharedMemory.addVote(true, MOTOR_ID,
                        decision.getId());
26              } else {
27                  switch (decision.getCommand) {
28                      case COMMAND_ONE:
29                          agree = states.get(2) && (
                                states.get(3) || states.
                                get(4));
30                          break;
31                      case COMMAND_TWO:
32                          agree = states.get(0) && (
                                states.get(1) || states.
                                get(2));
33                          break;
34                      default:
35                          agree = true;
36                          break;
37                  }
38                  sharedMemory.addVote(agree, MOTOR_ID,
                        decision.getId());
39              }
40          }
41      }
42
43      boolean rulesGeneration() {
44          return createRule(this);
45      }
46  }
```

Listing 4.3 – Pseudo code Java d'une règle de vote

La deuxième est la règle permettant la récupération des votes et la prise de décision. Cette règle est utilisée par un moteur qui possède une décision

au sein de la mémoire partagée qu'il n'a pas encore fermée (*closed* contient *false*).

Une fois qu'un moteur initie une décision, cette règle va s'exécuter. À chaque itération, en fonction du nombre de moteurs connectés à la mémoire partagée, la règle regarde si le nombre de votes pour la décision est égal au nombre de moteurs. Cependant, pour éviter que la règle ne boucle indéfiniment et que l'on tombe dans un *deadlock*, il existe également un *timeout* permettant de stopper l'attente après un certain moment défini.

Ce *timeout* permet de gérer efficacement un type de défaillances, les *stopping failures* détaillées à la section 3.4.1. Dans le cas où un processus s'arrête de fonctionner brutalement et indéfiniment, si le système fonctionne sur le principe de l'accord distribué, il ne faut pas attendre sa décision à l'infini, mais détecter sa défaillance avant.
Si après un certain temps, il manque des réponses de la part de certains moteurs, c'est que ces derniers ont été victimes d'une défaillance et qu'il faut prendre la décision finale sans eux.

Détaillons l'exécution de ce type de règle :

Tout d'abord, par le biais de la méthode *activationGuards()*, la règle vérifie qu'il existe une décision associée au moteur et si c'est le cas, elle vérifie que celle-ci n'est pas *closed*.
Si ces deux conditions sont vérifiées, le corps de la règle est exécuté.

- La règle récupère la dernière décision. Celle-ci, grâce à la vérification de la méthode *activationGuards()*, n'est pas terminée.
- Ensuite, elle récupère tous les votes de la décision grâce à son identifiant.
- Par une suite de conjonctions de la valeur *agree* de chaque vote, la règle calcule l'accord « global ».
- Deux choses peuvent ensuite déclencher la prise de décision :
  1. Soit le nombre de votes récoltés est égal au nombre de moteurs devant voter.
  2. Soit la différence entre l'heure actuelle et l'heure à laquelle a été placée la décision est strictement supérieure au *timeout* mis en place.
- Si un des deux cas déclenche la prise de décision, la règle exécute la méthode correspondante à l'ordre en lui passant en argument le booléen représentant l'accord « global ». S'il vaut *true*, c'est qu'il n'y a pas de conflit et donc l'ordre peut être exécuté sans problème. Dans l'autre

cas, soit le moteur attend et relance une décision lors d'une itération future, soit il exécute une action secondaire prévue et qui ne pose pas de conflit.

Comme pour la règle des votes détaillée au paragraphe précédent, le pseudo-code Java de ce type de règle se trouve ci-dessous.

```
class HandleDecisions {

    final long timeout = 30000L;

    void dataAcquisition() {
        sharedMemory = getSharedMemoryWrapper();
    }

    boolean activationGuards() {
        return
            !sharedMemory.isLastMotorDecisionClosed(
               MOTOR_ID) &&
            sharedMemory.getLastMotorDecision(
               MOTOR_ID) != null;
    }

    boolean conditions() {
        return true;
    }

    void actions() {
        decision = sharedMemory.getLastMotorDecision(
           MOTOR_ID);

        votes = sharedMemory.getVotes(decision.getId
           ());

        agreement = true;

        for (vote : votes) {
                agreement = agreement && vote.isAgree
                   ();
            }

        if (votes.size() >= sharedMemory.getNbrMotors
           () ||
```

```
31              (currentTime - decision.getTimestamp) >
                   timeout) {
32              switch (decision.getCommand) {
33                  case COMMAND_ONE:
34                      commandOne(decision.getId(),
                           agreement);
35                      break;
36                  default:
37                      break;
38              }
39              sharedMemory.closeDecision(decision.getId
                   ());
40          }
41      }
42
43      boolean rulesGeneration() {
44          return createRule(this);
45      }
46
47     void commandOne(agreement) {
48          if (agreement) {
49              action();
50          } else {
51              action_bis();
52          }
53      }
54  }
```

Listing 4.4 – Pseudo code Java d'une règle de prise de décision

4.4 Évaluation

Le chapitre 4 présente une solution quant à la problématique de la section 4.1.1. Cette solution reste assez simple et spécifique à la mise en réseau de moteurs proactifs, mais le but recherché n'était pas de développer une solution adaptable à tous les systèmes logiciels. Cette implémentation a permis de montrer qu'il est possible de mettre en réseau des moteurs proactifs, mais que cela génère plusieurs problèmes. Cependant, ces derniers peuvent être solutionnés par des mécanismes permettant de rendre le système plus robuste.

Le moyen de communication choisi est la mémoire partagée. Contrairement à l'échange de messages, cette dernière ralentit moins l'exécution du

moteur proactif. Chaque moteur s'occupe d'écrire ses données si et seulement s'il y a eu un changement par rapport à l'itération précédente. Comme détaillé dans le chapitre 4 de la solution, ceci peut être intégré sous forme de règle, ce qui rend l'ajout entièrement transparent vis-à-vis de l'exécution du moteur. En adoptant une communication par échange des messages, plusieurs choses auraient dû être mises en place comme l'envoi des messages, un *buffer* pour la réception simultanée, le traitement de messages de manière asynchrone, de la retransmission, etc. Toutes ces choses auraient également modifié l'exécution du moteur afin de l'adapter à la transmission de messages.

Concernant le nombre de messages, chaque moteur écrit ses données partagées en mémoire directement. Il y a donc tout au plus un message par source de données par moteur et par itération. Admettons n le nombre de sources de données et 1 le nombre de destinataires (la mémoire partagée), par itération le nombre maximum de messages envoyés par un moteur est (n * 1) ou n.
Par échange de messages, chaque donnée partagée doit être transmise à chaque autre moteur du réseau, ce qui multiplie le nombre de messages envoyés par un moteur à chaque itération. Admettons n le nombre de sources de données et m le nombre de moteurs, par itération le nombre maximum de messages envoyés par un moteur est (n * m).

La prise de décision pouvait prendre plusieurs formes dont l'atteinte d'un consensus et l'élection d'un *leader*. Afin de prendre les décisions de manière entièrement décentralisée, l'atteinte d'un consensus a été choisie. Cela permet à chaque moteur du réseau d'être sur un pied d'égalité en termes de décision et qu'un d'entre eux n'ait pas plus de « pouvoir » dans la décision finale. Si le choix avait été d'élire un *leader* permettant de centraliser, à la fois communication et prise de décision sur un même nœud, le moteur et son exécution auraient eu besoin de quelques adaptations. En outre, la base de données d'un moteur n'est adaptée que pour du stockage local, elle ne possède aucune communication avec l'extérieur permettant de recevoir les données de chaque moteur et de les stocker.

Les *deadlocks* ou situations d'interblocage sont à éviter. Quand elles se produisent, le système est entièrement bloqué et son exécution ne se poursuit plus comme prévu initialement. La mise en réseau peut faire survenir de telles situations et il est important de mettre des mécanismes en place afin de les éviter. La mise à jour de la mémoire partagée est indépendante puisque chaque moteur en possède la responsabilité. Si l'échange de messages avait été implémenté, des situations auraient pu se produire en perdant certains messages définitivement par exemple.

Le système distribué peut rencontrer un certain nombre de défaillances

pendant son exécution. Le *timeout* dans la prise de décision et le champ *timestamp* de chaque source de données en mémoire partagée permettent de prévenir les défaillances définitives (*stopping failures*). Cependant, ces mécanismes sont impuissants pour contrer les *byzantine failures*. Les moteurs vont voir que les données sont à jour, mais les valeurs aberrantes seront indétectables. Les défaillances de communication seront détectées grâce aux mêmes mécanismes que pour les défaillances définitives, mais aucune retransmission n'est mise en place afin de réparer la communication éventuellement défaillante.

La mémoire partagée pose également un problème majeur, celui de la centralisation des données. À l'inverse de la prise de décision qui est décentralisée, la mémoire partagée est identifiée comme un *single point of failure*. Si cette dernière rencontre un problème, la gestion de conflits comme la communication entre moteurs n'est plus opérationnelle. Afin d'éviter cette situation, la mémoire partagée peut être répliquée en permanence à un autre endroit en étant prête à prendre le relais si un problème venait à survenir.

La prise de décision dans les systèmes logiciels est soumise à des règles strictes. Il est important de connaître le chemin « parcouru » dans le système qui nous conduits à telle ou telle décision. L'algorithme de vote construit dans le chapitre 4 permet une traçabilité faible, car il est uniquement possible de savoir quel moteur a initié une décision, à quelle heure et le type de cette dernière. On connaît également le vote éventuel de chaque moteur accompagné de l'heure de celui-ci. Cependant, il est impossible de connaître l'heure de fin de la prise de décision pour détecter un éventuel *deadlock* ou encore de connaître les raisons d'un vote pour ou contre une décision de la part d'un moteur (connaître l'état dans lequel il était lors de son vote). Ces points sont des choses qui peuvent être améliorées pour rendre le système plus abouti et plus robuste.

Chapitre 5

Application à des cas réels

5.1 Contexte

La solution développée dans le chapitre 4 permet de rendre possible tout un tas d'applications concrètes qu'il était plus difficile de mettre en place auparavant. Cette section synthétise le prototype développé lors du stage effectué à l'Université du Luxembourg durant l'année académique 2017-2018. Le prototype développé s'est concentré sur la mise en réseau et la prise de décision entre plusieurs moteurs proactifs répartis dans une maison connectée truffée de capteurs et actionneurs. Il a également été question de configurer un ensemble de scénarios pour chaque moteur afin de contrôler et piloter la maison de façon la plus optimale possible.

Les moteurs proactifs embarqués de la section 2.3.2.2 peuvent être installés sur le smartphone des habitants de la maison afin d'y stocker leurs différentes préférences. Ces préférences vont altérer le comportement des scénarios configurés dès que le moteur embarqué est détecté au sein de la maison, une fois en dehors ces préférences ne sont plus prises en compte.
Ces moteurs vont devoir communiquer avec la maison connectée afin de communiquer les préférences qu'ils contiennent.

5.2 Maisons connectées

Les maisons connectées peuvent agir sur plusieurs domaines de la vie quotidienne principalement.

Conformément à leur définition, les maisons connectées sont connectées à internet. Cette connexion permet, entre autres, de les contrôler à distance. Par exemple, un utilisateur de celle-ci peut surveiller la température ambiante de la maison à un certain moment et si celle-ci ne nous convient pas,

il peut la modifier selon ses préférences.

Une autre caractéristique des maisons connectées est l'automatisation des tâches. La domotique permet aux tâches d'être automatisées et en les programmant afin de se produire à des instants précis, la maison devient entièrement autonome pour autant que le matériel présent le permette. Par exemple, les volets peuvent s'ouvrir en semaine à 7h et le week-end à 9h pour se fermer tous les jours à 20h.

Ce ne sont pas les seules caractéristiques d'une maison connectée, car l'éventail des possibilités est large. Mais ces quelques exemples couplés à l'émergence de l'internet des objets permettent d'apercevoir les choses qu'il est possible d'effectuer grâce à une maison devenue connectée.

5.3 Prototype

Lors du stage à l'Université du Luxembourg, la réalisation d'un prototype entièrement fonctionnel a été demandée. Au vu du contexte posé(section 5.1), l'environnement des maisons connectées (section 5.2) fut un bon choix.

5.3.1 Description

Le prototype a été défini comme suit :

- **Une maison connectée :** Munie de multiples capteurs et actionneurs placés à des endroits totalement différents les uns des autres en raison de divers environnements, la maison se veut complètement autonome.
  - **Capteurs :** Des capteurs de plusieurs sortes sont installés dans la maison : luminosité, mouvements, taux de CO, taux de CO_2, taux d'humidité, température ambiante, température extérieure et température de l'eau.
  - **Actionneurs :** Il existe aussi divers actionneurs : alarme incendie, alarme monoxyde de carbone, alarme de sécurité, chaudière, climatisation, fenêtres, lumières, portes, radiateurs et volets.
- **Des moteurs proactifs :** La maison décrite brièvement précédemment possède de nombreux senseurs répartis. Afin de rapprocher au plus près les moteurs proactifs de ceux-ci, plusieurs moteurs ont été intégrés. Chacun d'entre eux est entièrement indépendant vis-à-vis de ses senseurs et des données de ceux-ci.

- **Des moteurs fixes :** La maison possède trois moteurs dits « fixes ». Ils sont appelés de la sorte, car ils sont attachés à leurs senseurs et ne se déplacent pas dans l'espace. Chacun d'entre eux est entièrement indépendant, il est le seul à avoir accès à sa base de données contenant les données récoltées par ses capteurs.
- **Des moteurs embarqués :** En plus des moteurs « fixes », il existe des moteurs dits « embarqués ». Ceux-ci sont des moteurs proactifs plus légers et capables d'être exécutés sur un smartphone. Ils permettent le stockage de préférences pour chaque utilisateur et l'activation/désactivation de celles-ci lors de leur entrée/sortie au sein de la maison.

Chaque moteur « fixe », grâce aux données de ses capteurs, actionne ou non ses actionneurs afin d'exécuter ses scénarios prédéfinis.
Par la suite, grâce à la communication, chaque moteur va pouvoir prendre en considération l'environnement global et l'état des senseurs qu'il n'a pas à sa charge.
Enfin, grâce à la prise de décision, les moteurs dans leur ensemble vont pouvoir éviter et/ou solutionner les conflits qui pourraient survenir.

5.3.2 Composants

Afin de décrire les méandres du prototype, tous les composants ainsi que leurs interactions ont été décrits dans un modèle logique. Ce dernier est représenté à la figure 5.1.

La maison connectée possède trois moteurs proactifs. Chacun d'entre eux possède une base de données MySQL « locale ». Elle sert dans un premier temps à stocker les règles en cours d'exécution comme décrit à la section 2.3.1.2. Cette base de données étant propre au moteur, les capteurs et actionneurs y ont été connectés afin de stocker leurs valeurs et leurs états en temps réel.

Ce moteur proactif a également accès à une mémoire partagée. Cette dernière peut y accéder en utilisant l'API mise à sa disposition et contenant les méthodes associées.

Une dernière chose est l'ajout d'un moteur embarqué sur les smartphones. Un moteur est présent sur le smartphone des habitants et ce dernier contient leurs préférences. Ces moteurs doivent communiquer avec la maison connectée afin de l'informer des préférences à prendre en compte ou à désactiver en fonction du va-et-vient des habitants.

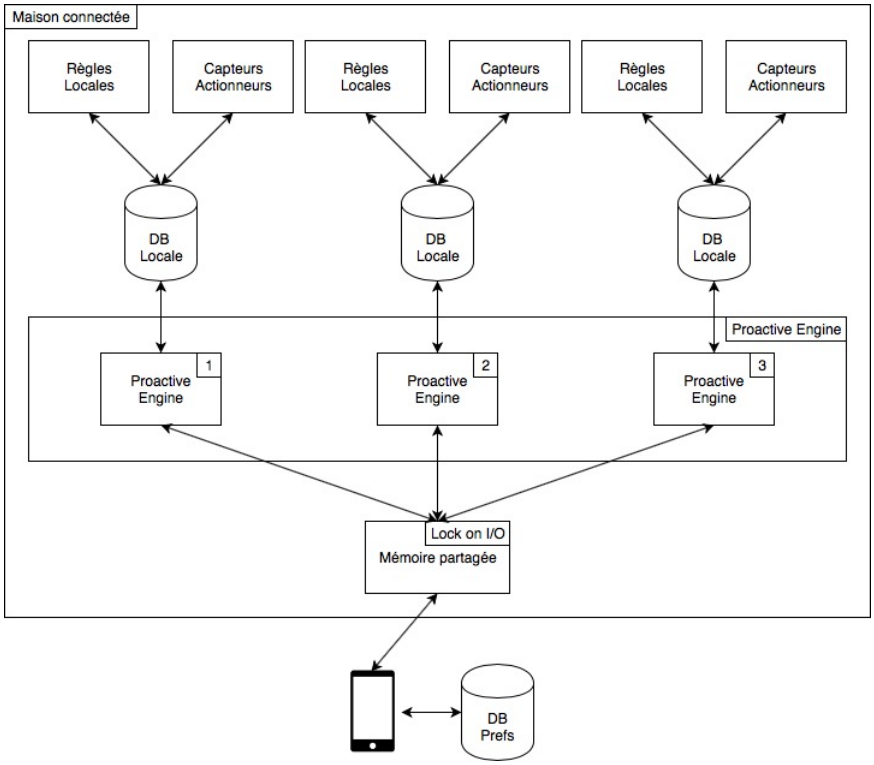

FIGURE 5.1 – Modèle logique représentant le prototype

5.4 Objectifs

Deux objectifs majeurs ont été conclus avec la conception du prototype :

— L'implémentation d'une communication sous forme de mémoire partagée afin de partager des informations entre chaque moteur.
— L'implémentation d'une prise de décision entre moteurs proactifs.

5.5 Apport du moteur proactif

Cette section apporte des arguments quant à la conception de ce prototype avec un moteur proactif plutôt qu'une autre technologie. Plusieurs choses peuvent être considérées comme un réel apport par ajout du moteur proactif à une maison connectée.

La première chose est la prise de décision autonome. Certes, dans ce cas-ci, il est toujours nécessaire d'avoir une action humaine, mais uniquement

lors de la configuration. Une fois les règles programmées, tout ce qui est exécuté sur le moteur devient entièrement autonome.

Deuxièmement, l'ajout de préférences. L'utilisateur enregistre ses préférences sur son smartphone. Une fois fait, le moteur s'occupe de tout et décide pour lui sans action externe visible de la part de l'utilisateur. Encore une fois, l'action humaine n'est nécessaire que lors de l'ajout ou la modification des préférences, mais on se détache d'une interface de configuration présente au sein de la maison.

Ensuite, il y a la résolution de conflits que certaines décisions peuvent provoquer. La maison devenant plus indépendante, elle est capable de raisonner sur son environnement ce qui la rend vulnérable à des décisions contradictoires. C'est pourquoi le moteur doit être capable de gérer les conflits et les solutionner afin de prendre des décisions cohérentes et logiques.

Et enfin, de l'optimisation peut être ajoutée. Le moteur sachant solutionner les conflits de façon autonome, il peut le faire de façon à prendre des décisions optimales au sein de la maison. Ordonnancer les choix, prendre un choix plutôt qu'un autre, tout cela afin d'atteindre un optimum préalablement défini.

Ces apports sont un avantage, car ils viennent « casser » la dépendance entre l'interaction de l'homme sur la machine. L'humain devient un superviseur n'ayant besoin d'intervenir qu'au début ou en cas de problème.

5.6 Scénarios

Afin de montrer la faisabilité de la solution du chapitre 4, plusieurs scénarios ont été configurés afin de contrôler la maison grâce aux senseurs présents. Ils ont été répartis équitablement dans les trois moteurs en fonction de la localisation des senseurs.

Il existe plusieurs types de scénarios comme : « Augmentation ou diminution de la température ambiante, allumage de l'alarme incendie, allumage de l'alarme de monoxyde de carbone, augmentation de la température de l'eau de la chaudière ... ».

Dans cette section, le scénario le plus complet va être détaillé. Les autres se basent sur les mêmes procédés, mais celui qui va être présenté contient tous les cas développés par la solution.

Le scénario choisi est celui de la diminution de la température ambiante au sein de la maison. Les seules données dont dispose le scénario sont celles récoltées par le moteur qui l'exécute, le reste donc être récupéré en mémoire partagée.

Le scénario « de base » (figure 5.2) commence par une règle « point d'entrée » qui vérifie la température de la maison en permanence. Une fois le seuil fixé dépassé, le scénario se met en marche. L'action prévue en bout de course est l'allumage de la climatisation.

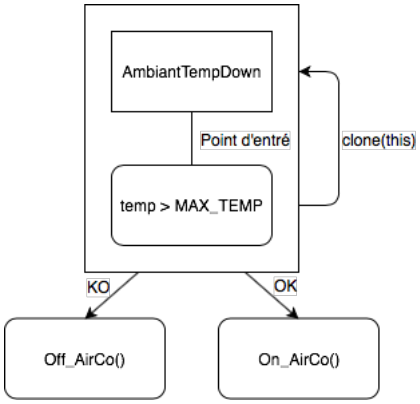

FIGURE 5.2 – Scénario basique contrôlant la baisse de la température

Plusieurs règles placées « au milieu » du scénario permettent quelques optimisations (figure 5.3). Le scénario s'assure que la maison est occupée et qu'il y a du mouvement grâce à la communication mise en place. Il va chercher toutes ces données en mémoire partagée, car elles proviennent d'autres capteurs attachés à d'autres moteurs.

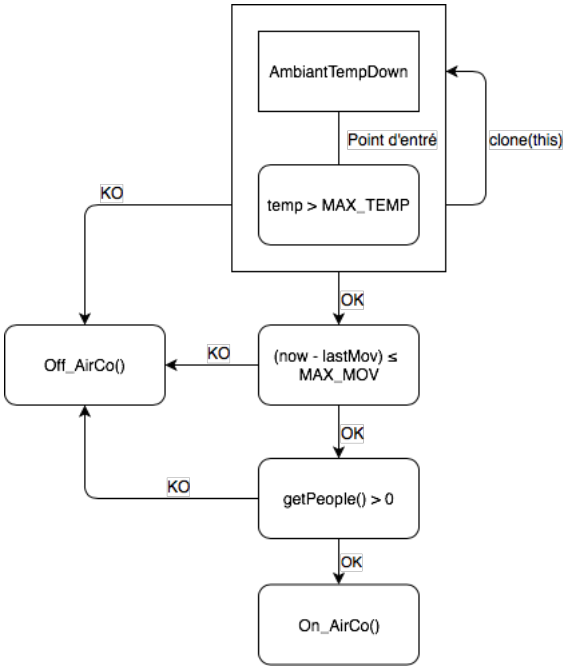

FIGURE 5.3 – Scénario contrôlant la baisse de la température (1er optimisation)

Dans cette version du scénario, un premier conflit est détecté. La maison connectée possède des fenêtres qui sont actionnées par les moteurs également. Or, si les fenêtres sont ouvertes pour une quelconque raison, il est aberrant d'actionner l'air conditionné. C'est ici que la détection de conflits de la section 4.3.3 va prendre tout son sens (figure 5.4). En plus des vérifications de données à des fins d'optimisations, il faut désormais vérifier que les fenêtres ne sont pas ouvertes s'il faut actionner la climatisation. Cette vérification se fait par le biais de la mémoire partagée, car les fenêtres sont sous le contrôle d'un autre moteur.

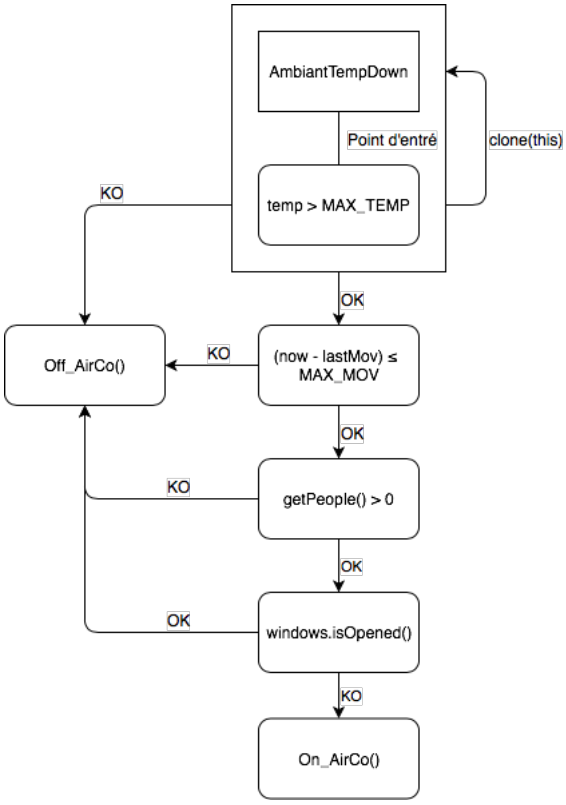

FIGURE 5.4 – Scénario contrôlant la baisse de la température (détection de conflits)

Grâce à cette communication, le scénario peut optimiser ses choix de façon plus poussés (figure 5.5). Au lieu d'actionner la climatisation si les conditions sont vérifiées, il peut vérifier la température extérieure et adapter son comportement en fonction du résultat. Si la température extérieure est inférieure à la température ambiante actuelle, la maison ouvre ses fenêtres (optimisation). Dans le cas contraire, elle conserve le comportement actuel, c'est-à-dire l'ouverture de l'air conditionné. Les fenêtres étant sous le contrôle d'un autre moteur, cette action prend la forme d'un ordre écrit directement en mémoire partagée. La détection de conflits peut encore être utilisée, la maison connectée possède des volets automatisés et ceux-ci, s'ils sont fermés, bloquent l'ouverture des fenêtres.

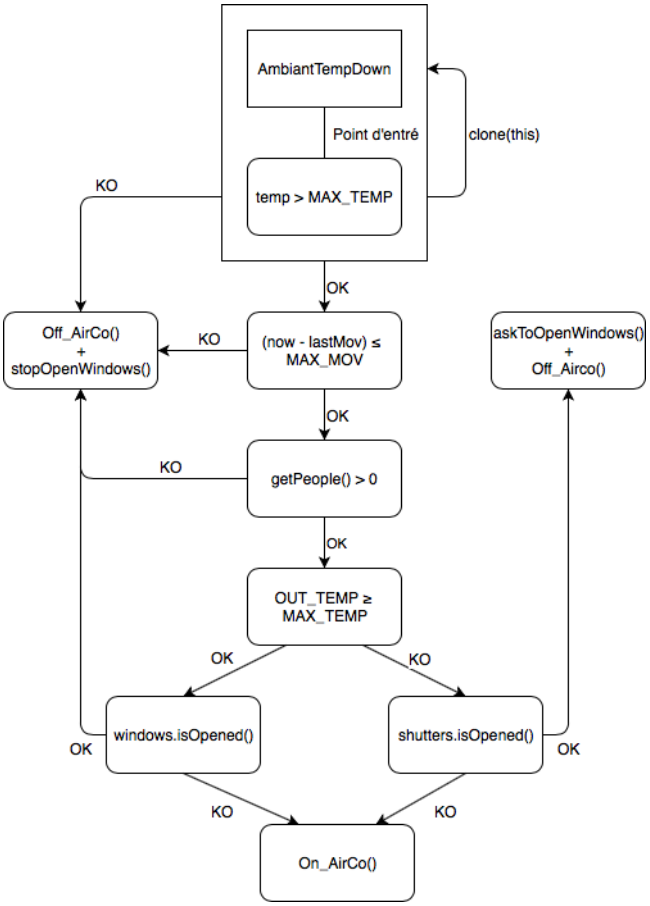

FIGURE 5.5 – Scénario contrôlant la baisse de la température (détection de conflits + passage d'ordre)

Pour terminer cette section, la résolution de conflits va être utilisée. Le moteur exécutant ce scénario peut demander, via la mémoire partagée, à un autre moteur l'ouverture des fenêtres. Cette action n'étant pas prévue « de base » au sein de ses scénarios, il ne connaît pas avec exactitude les valeurs à vérifier pour éviter un conflit. En conséquence, le moteur ayant la responsabilité d'actionner les fenêtres initie un vote dès que l'ordre se trouve en mémoire partagée. Grâce à ce dernier, il va pouvoir vérifier si des conflits existent et si cet ordre doit être exécuté ou simplement ignoré.

5.7 Préférences utilisateurs

Une des dernières choses présentes dans le modèle logique qui n'a pas encore été abordé est la gestion des préférences des utilisateurs.

5.7.1 Connexion et déconnexion

Avant d'entamer la gestion en elle-même, il est impératif de développer une solution de communication entre la maison connectée et les téléphones que possèdent les utilisateurs.

Une des solutions proposées qui a été retenue est l'envoi de « requêtes » par le moteur proactif embarqué.

En solution réelle, la maison connectée ne s'occupe pas des moteurs proactifs embarqués, elle agit uniquement de manière passive. La mémoire partagée ne se contente que de réceptionner les préférences d'un utilisateur.

Une fois un ensemble de préférences reçu, elle les stocke en mémoire afin que tous les moteurs « fixes » y aient accès. Ces préférences possèdent aussi un *timeout* envoyé par le smartphone.

C'est ce *timeout* qui va permettre à la maison connectée d'être passive. Si, une fois ce *timeout* dépassé, la mémoire partagée n'a pas reçu de mise à jour du moteur embarqué, elle peut considérer cet ensemble de préférences comme désactivé. Si le moteur a envoyé un signal, le *timeout* peut être allongé et les préférences restent actives.
Le moteur embarqué conserve également le *timeout*. Cela lui permet de savoir quand ses préférences deviennent obsolètes et quand il doit les envoyer de nouveau.

Les principaux avantages sont :

— Un utilisateur peut s'absenter quelques minutes en dehors de la portée de la maison tout en ne voyant pas ses préférences désactivées. Cependant, si le *timeout* se déclenche pendant son absence, le moteur embarqué devra signaler sa présence au retour.
— Si le *timeout* est de quelques minutes, l'utilisateur peut changer ses préférences afin qu'elles soient prises assez rapidement en compte par la mémoire partagée.

L'autre solution existante est celle de rendre la maison active. Dans la solution précédente, il s'agissait des moteurs embarqués qui avertissaient la maison connectée de leur présence.

La deuxième idée présente les choses dans l'autre sens. La maison connectée peut posséder un moteur proactif « fixe » supplémentaire qui ne supporte que la communication avec les moteurs sur smartphone.
Ce moteur « fixe » va scanner la maison à la recherche de moteurs embarqués qui n'auraient pas encore activé leurs préférences au sein de la mémoire partagée.

Une fois un nouveau moteur embarqué détecté, ses préférences sont « tirées » vers la mémoire partagée. Mais le moteur fixe ne scanne pas que les nouveaux moteurs, il se charge également de la désactivation des préférences dès qu'un utilisateur quitte la maison.

Une dernière solution peut être l'utilisation de la puce GPS du smartphone. En connaissant la localisation de la maison, une fois entré dans un rayon de x mètres, le smartphone cherche un moteur présent dans la maison afin d'initier le dialogue. Une fois en dehors de ce rayon prédéfini, le moteur embarqué se met « en pause ».
Si un moteur supplémentaire ne peut être ajouté, une fois présent dans le rayon, le moteur embarqué peut directement contacter la mémoire partagée. Pour cette option, il est impératif que l'API soit implémentée sur les moteurs embarqués et maintenue à jour.

5.7.2 Stockage des préférences

L'autre point de cette sous-section est la gestion/le stockage des préférences dans la mémoire partagée.

Tout comme pour le stockage des valeurs partagées, une table est créée afin de stocker toutes les préférences. Selon la convention définie, cette table se nomme : *SM_ Preferences*. Elle est représentée par la figure 5.6.

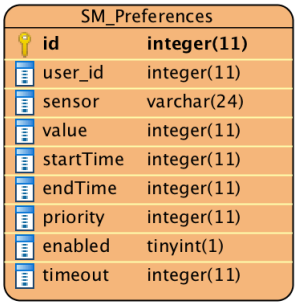

FIGURE 5.6 – Table regroupant les préférences utilisateurs

Chaque préférence représente une ligne et est identifiée par un identifiant. Une préférence comporte un *user_id* qui sert à identifier le smartphone de l'utilisateur. Un ensemble de préférences issu du même utilisateur porte le même « identifiant ».

La colonne *sensor* représente la fonctionnalité que cible la préférence. Par exemple, elle peut contenir « maxT » si la préférence concerne la température ambiante maximale. Bien sûr, il est possible d'ajouter n'importe quelle préférence pour autant que celle-ci soit prise en compte par les moteurs.
La colonne *value* contient la valeur de cette préférence. Par exemple, 22°C en reprenant l'exemple précédent.

Les colonnes *startTime* et *endTime* regroupent l'heure de début et de fin de chaque préférence. Chaque préférence est journalière, mais pour permettre une certaine modularité, une heure de début et de fin ont été ajoutées pour permettre d'avoir des préférences qui varient selon l'heure de la journée.
Les deux colonnes contiennent un entier qui indique le temps écoulé en minutes depuis minuit. Par exemple, 10h se traduit 600 pour 600 minutes après minuit. La contrainte sur ces deux colonnes est l'intervalle suivant : $[\![0\ ;\ 1439]\!]$.

La colonne *priority* désigne la priorité associée à cette préférence. Cette priorité, représentée par l'intervalle $[\![1\ ;\ 4]\!]$, sert à pondérer les préférences et à leur donner un poids plus ou moins fort dans le calcul qui sera abordé à la section suivante. Cette priorité, tout comme les préférences disponibles, peut être ajustée en fonction des besoins.

La colonne *timeout* désigne un entier qui contient le *timestamp* indiquant l'instant où les préférences doivent être désactivées si aucune mise à jour n'est reçue.

5.7.3 Calcul des préférences

Après avoir abordé le stockage des préférences, le calcul de celles-ci et leur intégration au sein du fonctionnement de la maison connectée vont être détaillés.

Comme détaillé dans le prototype de la section 5.3, il existe un moteur proactif « embarqué » qui nous permet d'ajouter les préférences utilisateurs sur le smartphone. Cette section va s'attarder sur la prise en considération des préférences utilisateurs contenues dans le smartphone des utilisateurs sachant que leur stockage en mémoire partagée et la communication avec celle-ci ont déjà été abordés dans la section 5.7.2.

Les préférences du prototype de la maison connectée se sont concentrées principalement sur quatre données différentes :

1. La température ambiante maximale
2. La température ambiante minimale
3. La luminosité minimale
4. L'humidité ambiante maximale

Afin d'optimiser l'intégration au sein du moteur, le calcul des préférences a été implémenté dans un scénario complet, composé de plusieurs règles. Ce dernier est représenté à la figure 5.7.

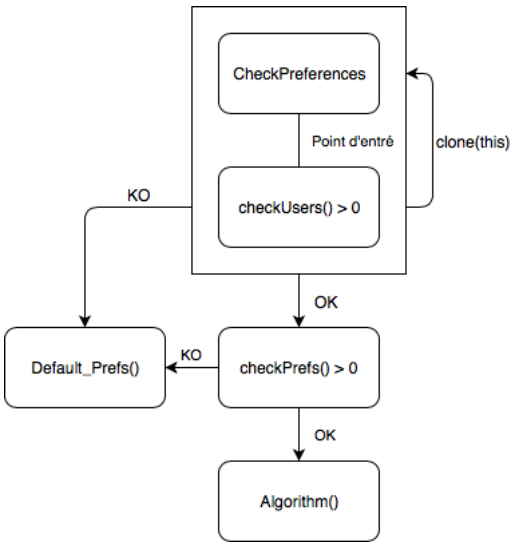

FIGURE 5.7 – Scénario du calcul des préférences

Avant de définir le point d'entrée, il est impératif de définir le cas « par défaut ». Cette règle se nomme *Default_ Prefs* ; elle permet d'appliquer des constantes de base pour chaque préférence définie si aucune d'entre elles ne peut être calculée. Lors de la configuration des scénarios, des valeurs par défaut pour les quatre données sont choisies afin d'être utilisées si aucune préférence n'est disponible.

La première règle, qui est aussi le point d'entrée, est *CheckUsers* mentionnée par la méthode *checkUsers()* dans la figure 5.7. Elle permet de vérifier

le nombre d'utilisateurs présents actuellement au sein de la maison. Si quelqu'un est présent, la règle récupère l'heure du système et la convertit en nombre de minutes écoulées depuis minuit. Mais si elle est vide, c'est la règle *Default_ Prefs* qui est exécutée afin d'utiliser les valeurs par défaut en remplacement des préférences.
La règle *CheckUsers* se clone afin d'effectuer sa vérification à chaque itération.

La deuxième règle est exécutée si *CheckUsers* vérifie que la maison n'est pas vide, il s'agit de *CheckPrefs* (appelée par la méthode *checkPrefs()* dans la figure 5.7). Des utilisateurs peuvent être présents au sein de la maison sans avoir défini de préférence pour l'heure actuelle.
En effet, dans la section 5.7.2 qui définit les préférences, celles-ci possèdent des heures de début et de fin et des créneaux horaires peuvent être vides de toute préférence.
Pour chaque préférence disponible et activée, la règle les prend en compte. Si pour une valeur, aucune préférence n'est disponible et activée, la règle applique la valeur par défaut pour cette préférence uniquement.
L'heure actuelle qui est utilisée par le système a été détaillée à la section 5.7.2.

La dernière chose qui compose le scénario est l'algorithme de calcul des préférences qui n'est pas une règle en lui-même.
Grâce à la règle *CheckPrefs*, pour chaque préférence, plusieurs choses ont été récupérées.

1. Le nom de la préférence qui représente le senseur impacté.
2. La valeur exacte actuelle
3. L'heure de début (formatée comme expliqué dans la section 5.7.2)
4. L'heure de fin (formatée comme expliqué dans la conversion dans la section 5.7.2)
5. La priorité de la préférence

Le calcul peut être séparé en deux catégories : les minimums et les maximums, bien que la solution d'une catégorie puisse être appliquée à l'autre.

Pour les valeurs maximales, il y a la température ambiante et l'humidité. Le principe est que la plus petite valeur de tous les maximums est acceptée, car la valeur maximale est vue comme un seuil à ne pas franchir. En effet, un utilisateur avec une valeur x et un autre avec une valeur y, si $x < y$, x est plus exigent. Comme y est plus tolérant et permet une valeur maximale plus haute que x, il sera également d'accord pour une valeur maximale plus basse qu'est x.

Pour chaque préférence impliquant un maximum, la plus petite valeur de toutes celles disponibles sera retenue.

Par exemple, pour la température ambiante maximale, trois utilisateurs ont choisi respectivement 26°C, 27°C et 28°C. En choisissant 26°C comme température maximale, les deux autres seront satisfaits puisque leur valeur maximale qu'ils avaient choisie n'a pas été dépassée. Or, en choisissant 27°C ou 28°C, un ou deux utilisateurs n'auraient pas été satisfaits en voyant leur limite dépassée.

Pour les valeurs minimales, il y a la température ambiante et la luminosité. Cette catégorie fait intervenir les priorités définies sur les préférences. Pour calculer une valeur minimum, le calcul se présente comme ceci :

Pour chaque type de préférence (par exemple, la température minimale), toutes les préférences associées sont récupérées.

- Tout d'abord, pour chaque préférence, la valeur est multipliée par la priorité, ce qui donne val_i.
- Tous les val_i d'un type de préférence sont additionnés, ce qui donne le résultat *totalVal*.
- Chaque priorité étant un chiffre, pour toutes les préférences d'un type, les priorités sont additionnées pour donner la valeur *sumPrio*.
- Pour finir, le calcul de la valeur minimum pour un type de préférence s'effectue grâce à la formule : (*totalVal* / *sumPrio*). La valeur obtenue est arrondie grâce à la fonction : *Math.round()*.

Par exemple, pour la température minimale, trois utilisateurs ont choisi 20°C, 21°C et 22°C. Respectivement, chaque utilisateur possède une priorité 1, 2 et 3. Le calcul de la température ambiante minimale se fait comme ceci : *Math.round(*((20 * 1) + (21 * 2) + (22 * 3)) / (1 + 2 + 3)*)* = 21,3°C.

Le prototype actuellement développé ne se concentre que sur quatre préférences à cause du nombre de senseurs disponibles dans la maison. Cependant, il est tout à fait envisageable d'en ajouter au vu de la solution existante.

Pour résumer, différentes approches existent.
En ne considérant pas de priorité entre les utilisateurs, la première solution peut être appliquée pour les deux catégories en choisissant la plus petite de toutes les valeurs dans le cas des maximas et la plus grande dans le cas des minimas. En ajoutant des priorités sur les préférences d'un utilisateur, la deuxième solution peut être utilisée pour pondérer les préférences tant à la hausse qu'à la baisse.

Chapitre 6

Conclusion

Nous avons commencé ce mémoire par situer le contexte historique du « proactive » et de l'« autonomic computing » dans le domaine informatique. Ce paradigme montre qu'il est impératif de casser l'interaction homme/machine pour faire face à la complexité grandissante des systèmes d'information. Cette rupture permet au système d'avoir une certaine autonomie en adoptant une dimension automatique de configuration, d'optimisation et de protection.

La suite de cet historique a été consacrée à l'implémentation de solutions intégrant de la proactivité. Grâce à celle-ci, les systèmes jouissent d'une fiabilité accrue due à la diminution de leur complexité. Le système tout particulièrement analysé a été le moteur proactif développé à l'Université du Luxembourg par l'équipe de Denis Zampunieris qui se base sur l'exécution de règles et de scénarios. Le point fort de cette solution est le niveau d'autonomie du système et cette distance qu'elle peut prendre avec l'interaction utilisateur.

Le domaine analysé après le « proactive computing » a été celui des algorithmes distribués. Face à l'émergence toujours plus grandissante de systèmes interconnectés, des algorithmes distribués sont impératifs. Nous avons analysé un bref historique de ceux-ci et nous avons présenté leur contexte d'application et les différentes caractéristiques qui font qu'un algorithme de ce type peut prendre différentes formes. Pour terminer ce chapitre, nous avons également présenté deux exemples d'application concrète de ceux-ci.

Après l'analyse de l'état de l'art, nous avons énoncé notre problématique. L'objectif fixé par celle-ci était de mettre en réseau plusieurs moteurs proactifs afin de pouvoir contrôler plusieurs systèmes. Le moteur proactif de M. Zampunieris possède les fonctionnalités nécessaires au contrôle d'un seul système via des scénarios proactifs. À cause de la problématique, nous avons relevé la nécessité d'ajouter des composants et plus précieusement un moyen

de communication à travers le réseau et une politique de management et d'ordonnancement des moteurs.

Parmi les composants additionnels abordés, le premier a été le moyen de communication entre les moteurs proactifs. Nous avons pris en considération deux voies qui sont l'échange de messages et la communication par mémoire partagée. Nous avons présenté les avantages et les inconvénients de chaque solution pour terminer par l'implémentation d'une mémoire partagée. Plusieurs éléments qui doivent être mis en place pour assurer son bon fonctionnement ont été analysés et détaillés. La mémoire partagée est un bon moyen de communication, car elle reste plus légère et plus fiable (grâce aux différents *timestamp*) que l'échange de messages. Cependant, il faut garder à l'esprit que cette méthode représente un *single point of failure* qui peut être critique pour certains systèmes distribués.

La problématique a également soulevé le fait qu'une politique de management devait être mise en place. En analysant les différentes possibilités à notre disposition pour résoudre le problème de l'accord, notre choix s'est dirigé vers la mise en place d'un consensus entre moteurs aux dépens de l'élection d'un *leader*. Afin de trouver un consensus, nous avons expliqué dans les détails un algorithme de vote s'inspirant de solutions existantes et comment ce dernier s'intégrait dans l'exécution actuelle du moteur proactif. En adoptant pour le consensus, nous avons opté pour un système décentralisé à l'inverse de l'élection d'un *leader* qui est centralisé. Cette « décentralisation » permet une répartition de la connaissance dans chaque moteur, car les données restent propres à chaque moteur et elle parfaitement adaptée à l'ajout ou la suppression de moteurs proactifs. En répartissant la logique, l'effort de maintenance à fournir peut s'avérer plus lourd que dans l'élection d'un *leader*. Un autre élément à prendre en compte est que l'attente des votes est fortement couplée à l'exécution du moteur. Ceci peut impacter la gestion des conflits en terme de temps de réaction par rapport à un évènement.

Nous avons terminé le mémoire par l'application de la solution à des cas réels en développant un prototype sur une maison connectée. Ce prototype montre qu'il est possible d'appliquer la solution à un environnement restreint, mais que l'implémentation peut également se faire dans un environnement plus large. Grâce à la réutilisation du moteur, nous avons pu contrôler plusieurs systèmes en même temps et de manière locale en développant les scénarios proactifs nécessaires. Cette application n'est que le début de ce qu'il est possible de faire avec un réseau de moteurs proactifs et elle donne des idées quant aux futures applications qui peuvent être mises en place avec ce système.

À travers ce mémoire, nous avons montré la possibilité de mettre en réseau plusieurs moteurs proactifs en ajoutant une dimension de communication ainsi qu'une politique de management permettant la gestion des conflits entre eux.

Pour terminer, plusieurs perspectives peuvent être envisagées pour un travail futur. La première est la mise en place de traces quant à la prise de décisions. L'algorithme de vote implémenté ne conserve que les décisions initiées et les votes des moteurs. À des fins de traçabilité, il peut être judicieux de conserver l'ensemble des traces d'une décision : initiation (identifiant du moteur, *timestamp*, état de moteur, conflits qui génèrent cette initiation), votes (identifiant des moteurs ayant voté, contenu du vote, *timestamp* du vote, état du moteur au moment du vote afin de comprendre le contenu de ce dernier) et prise de décision (état du moteur au moment de la prise de décision afin de comprendre son action). Une autre perspective est une amélioration de la mémoire partagée. Comme énoncé précédemment, cette dernière est un *single point of failure* et cela peut poser problème dans certains cas. Afin de combler ce problème potentiel, une réplication des données peut être envisagée. Quel que soit le moyen envisagé pour la copie, cela permettrait d'avoir un *back-up* à disposition sur lequel le système peut compter en cas de défaillance.

Bibliographie

[1] Pre Safe Assist - Volkswagen. `http://www.garagehaiti.com/fr/actualites/act-acc-presafe-assist-la-technologie-volkswagen`. Visité : 2018-05-06.

[2] PureStorage - Pure1 Support. `https://www.purestorage.com/fr/products/pure-1/support.html`. Visité : 2018-05-06.

[3] Technologies de l'information – techniques de sécurité – systèmes de management de la sécurité de l'information – vue d'ensemble et vocabulaire. ISO/IEC 27000 :2018, International Organization for Standardization and International Electrotechnical Commission, Geneva, Switzerland, 2018.

[4] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1) :87–90, 1958.

[5] Florian Daloze. La proactivité dans l'Internet des Objets : réponses en temps réel et optimisations. Master's thesis, Université de Namur, Faculté d'informatique, 2017.

[6] Edsger Wybe Dijkstra. *A short introduction to the art of programming*, volume 4. Technische Hogeschool Eindhoven Eindhoven, 1971.

[7] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, 2003.

[8] Ajay D Kshemkalyani and Mukesh Singhal. *Distributed computing : principles, algorithms, and systems*. Cambridge University Press, 2011.

[9] Joseph CR Licklider. Man-computer symbiosis. *IRE transactions on human factors in electronics*, (1) :4–11, 1960.

[10] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[11] Nicolas Mayer. Optimization of Local Proactive Engine Embedded Into Android-based Mobile Device. Master's thesis, Université du Luxembourg, Faculté des Sciences, de la Technologie et de la Communication, 2016.

[12] Gilles Neyens, Remus-Alexandru Dobrican, and Denis Zampunieris. Enhancing mobile devices with cooperative proactive computing. In *Proceedings of the 5th International Conference on Advanced Collaborative*

Networks, Systems and Applications (COLLA 2015), pages 1–9. IARIA, 2015.

[13] Denis Shirnin, Sandro Reis, and Denis Zampunieris. Design of proactive scenarios and rules for enhanced e-learning. In *Proceedings of the 4th International Conference on Computer Supported Education, Porto, Portugal 16-18 April, 2012*, pages 253–258. SciTePress–Science and Technology Publications, 2012.

[14] Gerard Tel. *Introduction to distributed algorithms*. Cambridge university press, 2000.

[15] David Tennenhouse. Proactive computing. *Communications of the ACM*, 43(5) :43–50, 2000.

[16] Roy Want, Trevor Pering, and David Tennenhouse. Comparing autonomic and proactive computing. *IBM Systems journal*, 42(1) :129–135, 2003.

[17] Denis Zampunieris. Implementation of a proactive learning management system. In *Proceedings of" E-Learn-World Conference on E-Learning in Corporate, Government, Healthcare & Higher Education"*, pages 3145–3151, 2006.

[18] Denis Zampunieris. Implementation of efficient proactive computing using lazy evaluation in a learning management system (extended version). *International Journal of Web-Based Learning and Teaching Technologies*, 3(1), 2008.